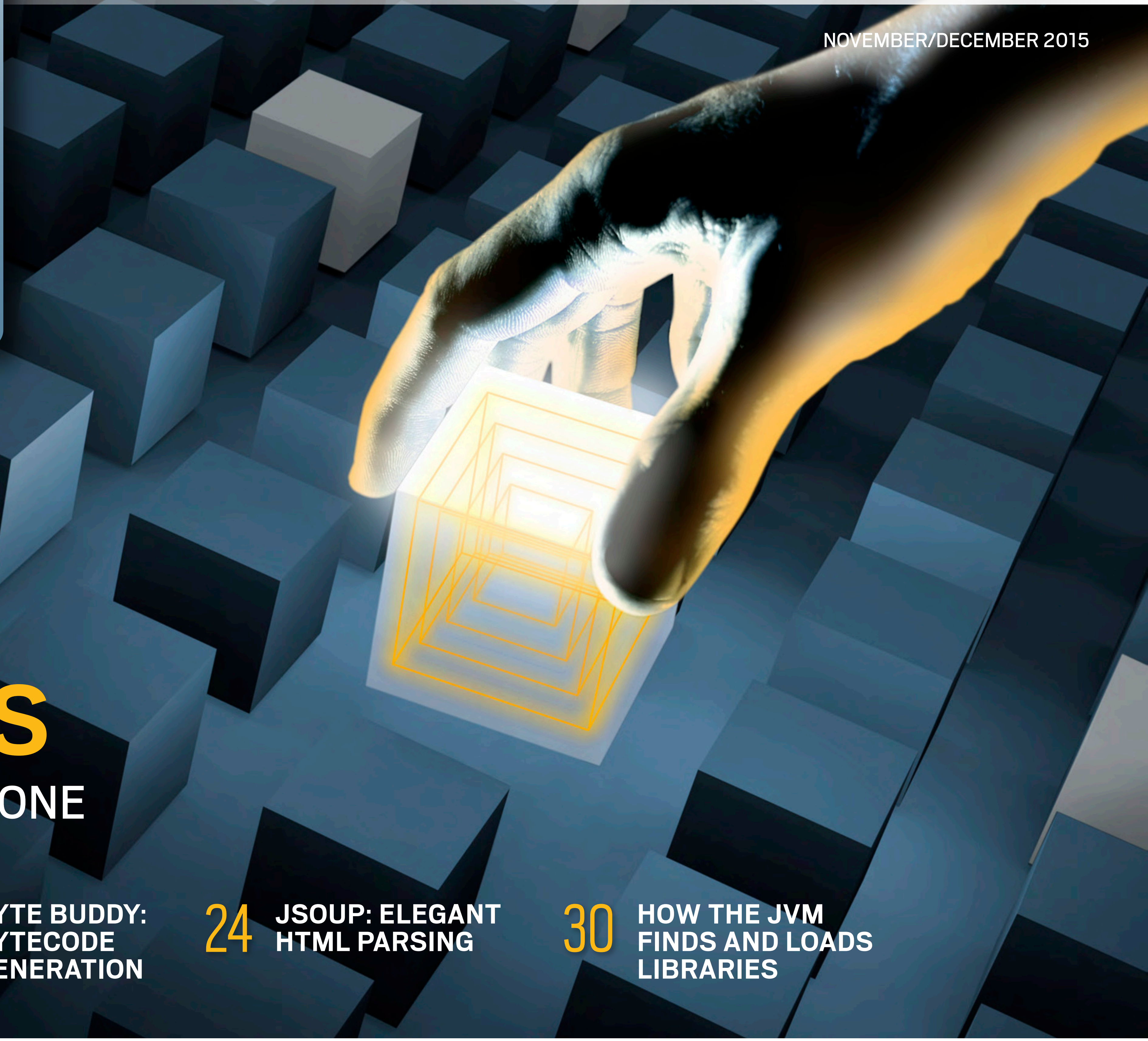
Java™ magazine

By and for the Java community

NOVEMBER/DECEMBER 2015



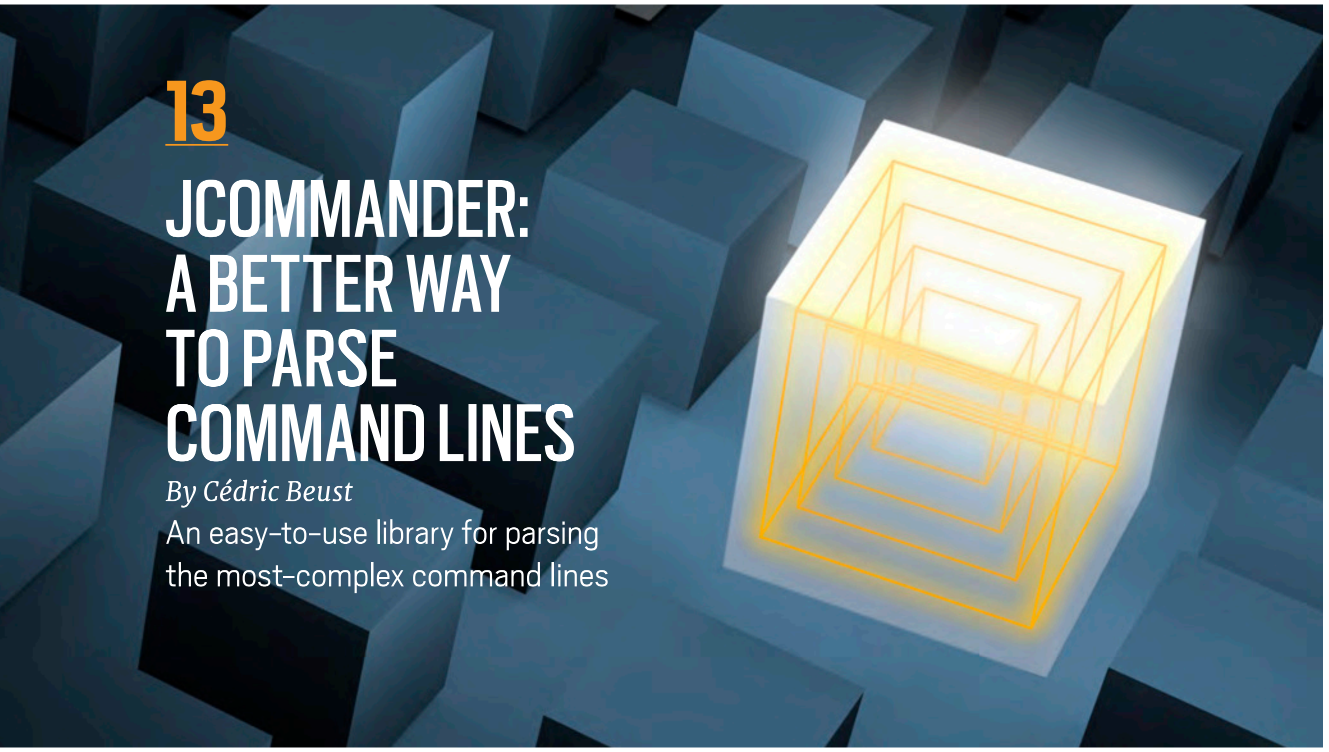
Libraries

FINDING THE RIGHT ONE

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /



COVER ART BY I-HUA CHEN

EDITORIAL
Editor in Chief
Andrew Binstock
Managing Editor
Claire Breen
Copy Editor
Karen Perkins
Section Development
Michelle Kovac
Technical Reviewers
Stephen Chin, Reza Rahman

DESIGN
Senior Creative Director
Francisco G Delgadillo
Design Director
Richard Merchán
Senior Designer
Arianna Pucherelli
Designer
Jaime Ferrand
Senior Production Manager
Sheila Brennan
Production Designer
Kathy Cygnarowicz

PUBLISHING
Publisher
Jennifer Hamilton +1.650.506.3794
Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985
Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES
Josie Damian +1.626.396.9400 x 200
Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201
Mailing-List Rentals
Contact your sales representative.

RESOURCES
Oracle Products
+1.800.367.8674 (US/Canada)
Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION
If you are interested in submitting an article, please e-mail the editors.

SUBSCRIPTION INFORMATION
Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE
java@halldata.com **Phone** +1.847.763.9635

PRIVACY
Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or e-mail address not be included in this program, contact Customer Service.

Java Magazine is published bimonthly at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.

Digital Publishing by GTxcel

XRebel

ZEROTURNAROUND

THE LIGHTWEIGHT JAVA PROFILER

TRY IT FREE NOW!

Get a free t-shirt!

BIO

Reforming Open Source Licensing

The complexity of open source licenses needs a remedy: the Creative Commons model.

In talking with developers and visiting the various forums we hang out in, I am constantly struck by how little is known about widely used open source licenses. While publishing code as open source in free, public repositories (a rapidly shrinking group of sites led by GitHub, Bitbucket, and SourceForge) continues to be a common way of sharing, the knowledge of what license to choose seems not to have moved forward. For many developers, the area of licensing is a large, ill-defined domain with weird terminology (*copyleft*, *BSD 2-clause*, and so on) that requires a lawyer to clarify. To save developers not interested in digging into the details in order to make their code available, certain mainstream licenses such as Apache, GPL, and MIT are generally recommended, with little useful explanation.

It might seem peculiar that intelligent individuals who delight in the smallest details of arguments on leaky abstractions will spend virtually no time understanding the basics of how open source software (OSS) licensing works. But to me, this view is backward: Why should a developer whose interests are precisely the details of the abstract constructs needed in his work be forced to understand the arcana of lawyer-written provisions in order to give his work to the larger community?

And the answer is that open source licenses are a rat's nest of overlapping and conflicting (and frequently complex) provisions born of historical accident. The licenses were never intended to make it easy for developers to understand. In part, this is due to omission by the group that oversees open source licenses,

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

the Open Source Initiative (OSI). The group was formed for various political purposes in the early days of open source and successfully did what was most needed at the time: end the proliferation of OSS licenses. Before the OSI, every company that released code would write its own license. While this was certainly in keeping with the idea of open source, it created the inevitable problem that it was very difficult to tell what exactly you could and could not do without reading through pages of legalese. By certifying certain licenses as "open source," the OSI helped stop the proliferation. And by encouraging the use of a subset of the approved licenses, it shrank the pool of licenses even further. (**Note:** However, you do not need the OSI's blessing to call your license or your code open source. In fact, several major projects today that we view as open source—for example, SQLite and TeX—are not issued under an OSI license.)

However, the pool of widely used OSI-approved licenses bulges with numerous provisions, many of which require an attorney to understand. Several OSS license analysts keep databases of the provisions. They count more than 500 different requirements in this pool of licenses. That number is daunting enough that it discourages developers from wading into them to understand the terms. As a result, the choice of license has become a nearly meaningless statement. But it shouldn't be.

Unless you recall the history of free software or have spent time studying terms of use, you might not know about the principal dividing line in these licenses, which maps to the key division between free software and open source. Generally, the former has a copyleft provision, which requires that any software that uses the licensed code be offered under the same license terms. (The Lesser General Public License, or LGPL, is the notable exception.) In counterpoint, OSS has no such requirement. So, choosing between GPL (free) and Apache License (OSS) is a meaningful action. Not meaningful, however, is BSD (3-clause) vs. BSD (2-clause). No one ever released code under one BSD license and later lamented that they hadn't chosen the other.

But without wasting time on websites, in forums, and with colleagues learning the intricacies, most developers would not know this. I strongly believe this problem can be solved by abandoning the use of licenses based on historical documents and moving to the model embraced by the Creative Commons, which is the OSI equivalent organization for artistic works.

The Creative Commons uses a series of graded licenses that start from public domain and allow for the addition of specific, defined requirements. These are few in number: acknowledgment of the creator, commercial use allowed or not, derivative works allowed or not, and requirements that derivatives be allowed only under the same license. With this system, an artist can tell right away which license to use without long study and analysis: she can mix and match the provisions and accurately choose the appropriate license.

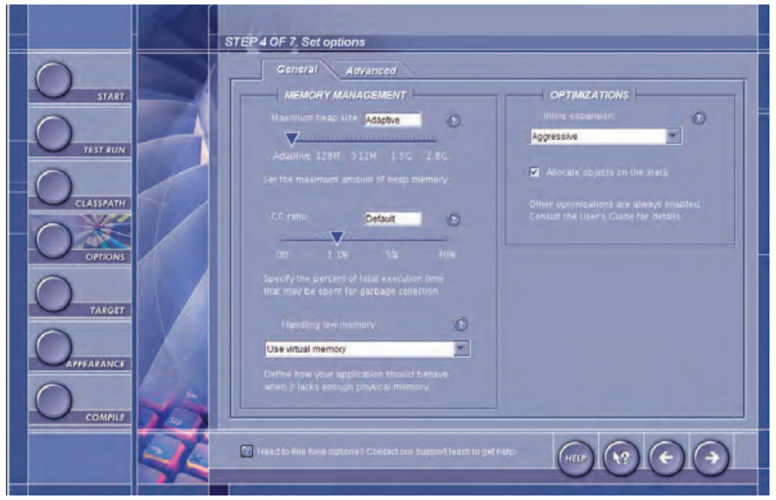
In software, this needs to be implemented—just a simple, sane system devoid of complex, competing, historical artifacts.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy

Why Compile Java Applications to Native Code Executables?

Since the early years of Java's two-decade history, production-grade JVMs relied on just-in-time (JIT) compilation for performance. A JIT compiler kicks in at application runtime and compiles the "hot" methods detected by the bytecode interpreter to native code.

The problem is that upon termination the JVM discards all code produced by the JIT compiler, which means it has to start over next time. The overhead of the interpret-profile-compile process, known as the "warm-up cycle," makes Java applications start more slowly, with slower initial response times compared to functionally equivalent native applications. This leads to the idea of ahead-of-time (AOT) compilation.



STEP 4 OF 7. Set options

MEMORY MANAGEMENT

Maximum heap size: Adaptive

Adaptive 128M 512M 1.5G 2.8G

Set the maximum amount of heap memory

GC ratio: Default

Off 1.1% 5% 10%

Specify the percent of total execution time that may be spent for garbage collection

Handling low memory: Use virtual memory

Define how your application should behave when it lacks enough physical memory

OPTIMIZATIONS

Inline expansion: Aggressive

☑ Allocate objects on the stack

Other optimizations are always enabled. Consult the User's Guide for details.

Need to fine tune options? Contact our support team to get help.

Excelsior JET Control Panel

An AOT compiler precompiles some or all application classes to native code before the application is run. That usually happens on the developer's system, although Android ART performs native compilation directly on the target device at application install time.

AOT compilation eliminates the warm-up cycle, enabling a large Java application to start 2 to 3 times faster and run at full speed from the start. It can also deliver an overall performance boost, especially in constrained environments with no spare computational resources or battery power for an advanced JIT compiler, such as embedded devices.

However, while faster startup and lower latency are valuable benefits, neither is the main motivation for using an AOT compiler when developing Java applications for desktop and server platforms.

The unique advantage of compiling the *entire* Java application to native code before deploying it in production or shipping to end users is actually *protection against Java decompilers.*

Class files emitted by the standard javac compiler are extremely easy to revert back to source—just search for "download java decompiler" and you will get your source code back in five minutes. Methods exist for making decompiler output less comprehensible, but they have serious limitations and may negatively affect application robustness and performance. Name obfuscation can only be applied to application classes that are not accessed via reflection or JNI, whereas references to the standard library classes remain visible. Excessive control flow obfuscation substantially hinders JIT compiler optimizations, penalizing application performance. Finally, software encryption does not protect your Java classes at all—they must be decrypted prior to execution, and dumping classes to disk as the JVM loads them is fairly trivial.

In contrast, a native binary produced by an optimizing AOT compiler carries no performance tax and is about as difficult to reverse-engineer as if you had coded the original program in C++. And you can still take additional measures to protect your intellectual property or enhance the security of your application if you wish: obfuscate names, encrypt strings and resources, and further protect the native code executable using platform-specific tools.

Excelsior JET 11 is the only certified Java SE 8 implementation capable of precompiling Java applications to optimized native code executables. It currently supports Windows, OS X, and Linux on Intel hardware.

For more information,
visit **www.ExcelsiorJET.com**

Java™ COMPATIBLE

EXCELSIOR

JULY/AUGUST 2015

Quizzical Quiz Questions

I enjoyed the July/August issue. However, I disagree with the answer to question 2 in Fix This. The article said the answer is B, but the answer is A. When the code executes, a single object is created on line 4, and at the end of the execution of the `main` method it is eligible for garbage collection because the created object is only ever referenced by local variables `e` and `e1`.

—Nathan Hawk (similarly Hans Yperman)

In Fix This for the July/August issue, question 3 contains an error. It refers to a code snippet in a file called Shop.java, which declares a class entitled `OnlineCart`. This is not legal in Java.

—Alexei Bolyrev (similarly Mushfiq Mammadov)

Editor Andrew Binstock responds: As mentioned in the introduction to this quiz, the questions were provided by the team that supports the Java certification exam publishing. After discussion with them, it came to light that the quiz questions had not been through the required technical review. The process has now been modified to ensure that only thoroughly reviewed quiz questions are used. Should that review not reach our standards of completely accurate content, we will switch to other sources to provide quiz questions. My apologies to all readers misled by these mistakes.

Java in the Real World

I read your review of Paul and Gail Anderson's book about JavaFX and the NetBeans Platform in the September/October issue. In commending the publishers for printing a book for "so narrow a topic," you ask, "How many such readers could there be?"

Are you asking how many applications are being created on the Java desktop? Or are you saying that JavaFX or the NetBeans Platform is a strange thing to use?

—Walter Nyland

Reviewer Andrew Binstock responds: I believe the community of developers who use NetBeans as a platform for their applications is small. Only a fraction of desktop applications need its capabilities, and those using JavaFX are a subset. So to see a nearly 900-page book aimed at them was, I felt, unusual and noteworthy.

Kindle Format

I noticed in the September/October issue that one of your subscribers was wishing he could read *Java Magazine* on his Kindle. That's very easy to do. This link shows how to do it.

—Brian Everett

Paper, Paper!

Concerning Jennifer Hamilton's reply in the September/October issue to Raj Thondepu, who inquired about a paper edition of *Java Magazine*: has Oracle actually researched this before concluding that a printed magazine wouldn't be sustainable?

—Martins O. Adegoke

Editor Andrew Binstock responds: Oracle publishes two magazines in hard copy, so it knows the numbers. It has just shifted one magazine to electronic-only and plans to eventually move the other. Paper is a very expensive medium that is suitable only for consumer magazines or for publications with high subscription prices.

Contact Us

We welcome comments, grumbles, and kudos at javamag_us@oracle.com. These might be edited for publication. If your note is private, please so indicate. For other ways to reach us, see the last page of this issue.

PHOTOGRAPH BY BOBI/GETTY IMAGES

Devoxx Belgium *NOVEMBER 9–13*
ANTWERP, BELGIUM
By developers for developers, this event has 200 speakers and 3,500 attendees from 40 countries. Tracks this year include Java SE, JVM languages, and server-side Java, as well as cloud and big data, mobile, and architecture and security, among others.

W-JAX 15
NOVEMBER 2–6
MUNICH, GERMANY
The W-JAX conference covers current and future-oriented technologies from Java, Scala, Android, and web technologies to agile development models and DevOps.

J-Fall 2015
NOVEMBER 5
EDE, NETHERLANDS
The annual Java conference organized by the Dutch Java User Group (NLJUG) typically sells out and has outgrown its usual venue. This year, J-Fall will take place in the CineMec in Ede.

Devoxx Morocco
NOVEMBER 16–18
CASABLANCA, MOROCCO
Formerly the JMaghreb conference, this event is a university day of training, workshops, and labs followed by conference days of sessions on software development, web, mobile, gaming, security, methodology, Internet of Things, and cloud. The Decision Makers evening includes discussion of issues related to the IT industry in Morocco.

QCon San Francisco 2015
NOVEMBER 16–20
SAN FRANCISCO, CALIFORNIA
A practitioner-driven software development conference, QCon is designed for technical team leads, architects, engineering directors, and project managers who influence innovation in their teams. Tracks this year include Taking Java to the Next Level and The Dark Side of Security. The last two days are devoted to workshops.

Codemotion Milan
NOVEMBER 18–21
MILAN, ITALY
This conference is open to users of all languages and platforms. It offers full-day workshops on the first two days, followed by keynotes and conference sessions.

Codemotion Spain
NOVEMBER 27–28
MADRID, SPAIN
This two-day event draws 2,000 attendees, represents more than 30 communities, and features coding lectures and workshops. Activities for startups, recruiting, and networking are included.

Clojure eXchange 2015
DECEMBER 3–4
LONDON, ENGLAND
Meet with the world's leading experts, learn how to use Clojure with your team, and discuss war stories with your peers. Both days will feature a mixture of talks covering various aspects of Clojure devel-

opment: from libraries to music, from ClojureScript to data.

Groovy and Grails eXchange 2015

DECEMBER 14–15
LONDON, ENGLAND
Stay ahead of the curve and hear the 2016 roadmap for Groovy and Grails from core committers and Groovy authorities Guillaume Laforge and Graeme Rocher. Engage with other leading experts and fellow enthusiasts and learn the latest innovations and practices.

Jfokus

FEBRUARY 8–10, 2016
STOCKHOLM, SWEDEN
Jfokus has run for eight years and is the largest annual Java developer conference in Sweden. Conference topics include Java SE and Java EE, front end and web, mobile, continuous delivery and DevOps, Internet of Things, cloud and big data, future and trends, alternative JVM languages, and agile development.

DevNexus 2016

FEBRUARY 15–17, 2016
ATLANTA, GEORGIA
DevNexus is a conference drawing 1,700 developers, with 6 workshops, 12 tracks, and 120 presentations. Featured tracks include HTML5 and JavaScript, Java SE/Java EE/Spring, and data and integration.

ConFoo 2016

FEBRUARY 22–26, 2016
MONTREAL, QUEBEC, CANADA
ConFoo is a multitechnology conference for web developers, featuring about 150 presentations by popular international speakers. Past sessions have included Testing Java EE Applications Using Arquillian by Reza Rahman and Hybrid Mobile Development with Apache Cordova and Java EE 7 by Ryan Cuprak.

Embedded World 2016

FEBRUARY 23–25, 2016
NUREMBERG, GERMANY
The 14th annual gathering of embedded system developers will explore the latest developments, define trends, and once again present the key areas of focus for future developments. This is where hardware, software, and system development engineers come together to turn the next milestones of the Internet of Things into reality.

Apache Hadoop Innovation Summit

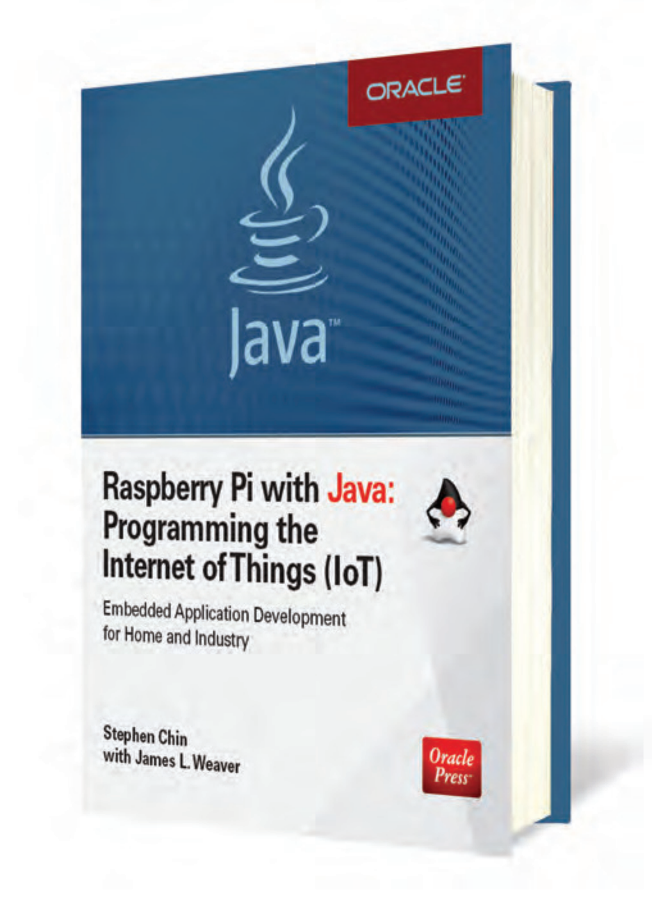
FEBRUARY 25–26, 2016
SAN DIEGO, CALIFORNIA
With presentations from more than 25 hands-on industry speakers, topics covered will include MapReduce and Spark, building privacy-protected data systems, scalable data curation, best practices, and architectural considerations for Hadoop applications.

Riga Dev Day

MARCH 2–4, 2016
RIGA, LATVIA
This event is a joint project by Google Developer Group Riga, Java User Group Latvia, and Oracle User Group Latvia. By and for software developers, Riga Dev Day focuses on 25 of the most-relevant topics and technologies for that audience. Tracks include JVM and web development, databases, DevOps, and case studies.

Have an upcoming conference you'd like to add to our listing? Send us a link and a description of your event at least four months in advance at javamag_us@oracle.com. We'll include as many as space permits.

PHOTOGRAPH BY TANGMAN PHOTOGRAPHY/GETTY IMAGES



RASPBERRY PI WITH JAVA: PROGRAMMING THE INTERNET OF THINGS (IOT)

By Stephen Chin and James Weaver
Oracle Press

Few can doubt that Java is an important language for the Internet of Things (IoT)—not only on the back-end servers that aggregate data from thousands of devices, but on the devices themselves. The Raspberry Pi, one of the most popular hardware platforms for developing IoT projects, has been able to run Java SE since 2013 (previously it could run OpenJDK). Since then Java SE has been bundled with the Raspbian image, which is the bootable OS image for the device.

Most projects built today on the Raspberry Pi are undertaken by hobbyists, in part because of its remarkable ease of use: only a modicum of knowledge is required to get up and running. The hardware is comparatively straightforward, and the key obstacles consist principally of understanding the interactions between the software and the hardware. This book is an introduction to this world for programmers who are already competent in Java programming but know little about the Raspberry Pi, much less how to program it.

The authors have written for *Java Magazine* in the past, and they take an informal, hands-on approach.

The first of the nine chapters takes you through the device itself and explains how it works. The second chapter is a detailed explanation of how to set up a Java programming environment using NetBeans. It then starts in with a simple, but not trivial, project: interacting with a scale and a thermocouple to make a perfect cup of coffee by measuring the beans and monitoring the water temperature. (This project was written up by the lead author in the May/June issue, which shows the congenial, informative style he uses.)

There follow projects that educate the reader on the basic use of the general-purpose input/output (GPIO) capabilities and other functions. These projects, like those that follow, contain notes for different versions of the Raspberry Pi and provide frequent links to external sites for obtaining software packages and hardware extensions. In many ways, they're a guide to the Raspberry Pi's ecosystem. Then, the authors get into projects that include interacting with RFID chips and building a robot, a drone control center, and a game console with Nintendo emulation.

My only complaint is that this book lacks an appendix containing reference information, so if you want to refer back to some hardware details, you need to remember in which project it was first presented. Other than this minor detail, this book is by far the best introduction I've seen to Java programming on the Raspberry Pi. *—Andrew Binstock*

Libraries:
FINDING NEW GEMS

Developers who recall the time when Java first came to prominence will remember that a key quest in those days was the concept of reusability. Object orientation had only recently come to the fore and there was considerable hope that objects might provide small, atomic components that could be stored in some sort of accessible repository for developers to share. The goal was to avoid having developers constantly write their own implementations of the common data structures, algorithms, and boilerplate. Programmers using C, with its minuscule standard library, surely recollect the pain point that reusability sought to address: getting out of writing yet another implementation of linked lists.

This original vision was fulfilled using a slightly less granular unit: the library. And today, a robust set of libraries is the hallmark of widely used languages such as C++, C#, Python, and others—including Java. In fact, Java has arguably the largest ecosystem of libraries to choose from.

This issue of *Java Magazine* focuses on several little-known libraries that emphasize ease of use for the developer: JCommander for handling the command line simply, despite supporting a wealth of complex options (page 13); Byte Buddy for generating or modifying bytecodes without requiring deep knowledge of bytecode syntax (page 19); and jsoup, a smartly designed Java library for parsing HTML (page 24). To round things out, we've included some rich nerd stimulation with a detailed article on how the JVM locates and loads libraries (page 30).

Did we miss a library we should have covered? Let us know. —*Andrew Binstock*

ART BY I-HUA CHEN

JCommander: A Better Way to Parse Command Lines

An easy-to-use library that exploits annotations to parse the most-complex command lines

CÉDRIC BEUST

BIO

A few years ago, I found myself needing to write an application that would be used mostly from the command line. It was a fairly ambitious project that would require complex command-line parameter parsing. So, naturally, my first instinct was to look for a library that would allow me to specify the command-line syntax of my application easily while remaining flexible. While I did find a few libraries for that purpose, they all struck me as being quite antiquated, using ideas and practices that even predate Java. In addition, they all failed to take advantage of Java's latest features.

So I started playing with a few ideas and next thing I knew, I had completely abandoned my initial idea and instead, I created JCommander: a modern, open source library designed to make it easy to parse command-line arguments while covering as many styles of argument syntax as possible. The arguments are not limited to strings, numbers, and commands, but can also include lists, arbitrary Java objects, passwords, and so forth. Let's have a look.

Quick Overview

The first realization I had when designing JCommander was that at the end of the day, once all the options on the command line have been parsed, the results end up in a Java object. Typically, it's a very simple object, referred to as a POJO (Plain Old Java Object), and it is often just a container with no logic methods in it—just fields with getters and setters.

Let's write a quick "hello world" program that allows us to parse the following line:

```
tool --name Cedric --verbose
```

We can capture the parsed information in the following class:

```
class Args {
      boolean verbose;
      String name;
}
```

To do this, we use annotations to tell JCommander how to initialize the class.

```
class Args {
      @Parameter(names = "--verbose")
      boolean verbose;

      @Parameter(names = "--name")
      String name;
}
```

Now, all we need to do is initialize JCommander with an instance of this class, pass it the command-line parameters and, once the parsing is done, the instance will contain all the right values, all properly assigned to the correct fields:

```
public static void main(String[] argv) {
    Args args = new Args();
    new JCommander(args).parse(argv);

    System.out.println("Hello " + args.name
    + ", verbose is: " + args.verbose)
}
```

Annotations are a very good match for this kind of approach for several reasons: the syntax is very cleanly laid out in the argument class, and just reading the source shows what the program accepts. There are also several other advantages that I will explain shortly.

The Power of Annotations

The most salient aspect of JCommander's approach is the use of annotations. I've always been a fan of Java's annotations, but I'm probably a bit biased because I was a member of the committee that designed them. Still, even now, ten years later, I continue to think they enable a style of programming in Java that's exceptionally expressive.

The important thing you need to remember about annotations is that they are a perfect match when you are trying to attach additional meaning to Java elements such as classes, fields, or methods. Any information that is not specifically tied to a Java element—such as package information, host names, or port names—should be specified externally. With this simple rule in mind, it's pretty clear that annotations are the right choice for JCommander.

The most salient aspect of JCommander's approach is the use of annotations. They enable a style of programming in Java that's exceptionally expressive.

In addition, annotations can have multiple attributes, which allows you to refine the metadata you are attaching to the Java elements. I showed only one attribute in the previous code, `names`, but here are a few others:

```
@Parameter(names = "{ --output, "-o" },
    required = true,
    description = "The output file")
String file;
```

Because I specified the `required` attribute, JCommander throws an exception if this parameter is omitted:

```
Exception in thread "main" com.beust.jcommander.Pa-
rameterException: The following option is required:
--output
```

Note that `names` is plural: you can specify multiple names for the attribute. This means the following two command lines are equivalent:

```
tool --out file
tool -o file
```

This capability addresses the common problem of users having different preferred styles for specifying command-line options.

Usage Explanation

I mentioned the attribute `description` previously because JCommander gives it special treatment: whenever a parameter has this attribute, it will automatically be collected and used to present a full description of the accepted syntax. If you ever want to display such a help message to users (for example, after they entered an invalid syntax), all you need to do is call `usage()` on your JCommander object and something like this will be displayed:

```
Usage: <main class> [options]
  Options:
    --debug          Debug mode (default: false)
  * --groups         Group names to be run
    --log, -verbose  Level of verbosity (default: 1)
    --long           A long number (default: 0)
```

This description includes as much information about the syntax as JCommander can gather based on your annotations of the options: their names, whether they are required (the asterisk denotes that), their default values and, of course, their descriptions.

This feature expresses another important principle in programming: don't repeat yourself. If you specified the syntax once in your argument class, you should not have to duplicate that effort if you want to display a help banner. JCommander takes care of this for you automatically.

Types

JCommander understands a lot of types by default, and all these types lead to the concept of *arity*. Arity defines how many values a parameter requires, for example:

- A boolean parameter needs no values: its value is true if it is present and false if it's omitted.
- A scalar (int, long, string, and so on) needs one value, for example, `--logLevel 3`.
- A list needs multiple values.

JCommander automatically infers these arities based on the type of your parameters. Additionally, if you are not satisfied with the default arities, you can define your own. This allows other kinds of syntax. For example, you can specify that a boolean has an arity of 1. This option would support syntax such as:

```
tool --verbose true
```

JCommander can support the most-complex codebases and syntax styles, so there are several features that help you organize your code cleanly.

JCommander even supports variable arities, which are parameters that can take any number of values, such as: `--files file1 file2 file3`.

These default types are sometimes not sufficient and your application might require even more-complex options to be specified. For example, we specified earlier an output file in the form of a string. Wouldn't it be convenient if instead, JCommander could deliver a real java.io.File object instead of a string? This is where type converters come in handy.

Let's modify our previous example to specify a real Java file instead of a string:

```
@Parameter(names = "-file",
    converter = FileConverter.class)
File file;
```

Note the additional `converter` attribute, which we need to implement:

```
public class FileConverter
    implements IStringConverter<File> {
  @Override
  public File convert(String value) {
    return new File(value);
  }
}
```

You can specify any number of type converters and JCommander will automatically use them based on the type of the field. It is that simple.

Syntax Flexibility

To support as many syntax styles as possible, JCommander allows you to specify separators other than spaces. For example, instead of `java Main -log 3` you might want to use `java Main -log=3` or `java Main -log:3`, which can all be specified with the `separators` attribute:

```
@Parameters(separators = "=")
public class SeparatorEqual {
  @Parameter(names = "-level")
  private Integer level = 2;
}
```

Validation

As your command grows complex, it will become tricky to decide whether a given command line is valid, because multiple options can interact with each other in various ways. JCommander can provide some assistance here by making it easy to validate your parameters. The syntax is very similar to the one we just covered with type converters:

```
@Parameter(names = "-age",
    validateWith = PositiveInteger.class)
private Integer age;
```

And here is the implementation of this validator:

```
public class PositiveInteger
    implements IParameterValidator {
 public void validate(String name, String value)
      throws ParameterException {
    int n = Integer.parseInt(value);
    if (n < 0) {
      throw new ParameterException(
        "Parameter " + name + " should be positive"
        + (found " + value +")");
    }
  }
}
```

Complex Commands

You might be familiar with a few tools that use subcommands to express sophisticated invocation syntax. For example, `git` offers this kind of syntax when multiple subcommands have their own syntax: if you call `git commit`, then you can specify parameters such as `--author` or `-amend`, while `git add` accepts `-i`. It's easy to implement this kind of syntax with JCommander.

First of all, you define your commands in their own classes. Here is `commit`:

```
@Parameters(separators = "=",
    commandDescription = "Record changes")
private class CommandCommit {

  @Parameter(description = "The list of files")
  private List<String> files;

  @Parameter(names = "--amend", description =
    "Amend")
  private Boolean amend = false;

  @Parameter(names = "--author")
  private String author;
}
```

And here is `add`:

```
@Parameters(commandDescription =
  "Add file to the index")
public class CommandAdd {

  @Parameter(description =
    "File patterns for the index")
  private List<String> patterns;

  @Parameter(names = "-i")
  private Boolean interactive = false;
}
```

Then, you add these commands to JCommander. In the following code, I have added a few assertions at the end to demonstrate what happens:

```
JCommander jc = new JCommander();

CommandAdd add = new CommandAdd();
jc.addCommand("add", add);
CommandCommit commit = new CommandCommit();
jc.addCommand("commit", commit);

jc.parse("-v", "commit", "--amend",
    "--author=cbeust", "A.java", "B.java");

Assert.assertTrue(cm.verbose);
Assert.assertEquals(jc.getParsedCommand(),
    "commit");
Assert.assertTrue(commit.amend);
Assert.assertEquals(commit.author, "cbeust");
Assert.assertEquals(commit.files,
    Arrays.asList("A.java", "B.java"));
```

As you can see from the assertions, the code has parsed the command line correctly and placed the arguments in the expected variables.

Architecture

JCommander can support the most-complex codebases and syntax styles, so there are several other features that help you organize your code cleanly.

Multiple argument objects. As your syntax grows, you might find yourself having one gigantic argument class that becomes a bit difficult to maintain. JCommander lets you break this class into multiple classes so that you can organize the options in a more intuitive way:

```
CommandRead argRead = new CommandRead();
CommandWrite argWrite = new CommandWrite()
JCommander jc = new JCommander(argRead, argWrite);
jc.parse(argv);

// argRead and argWrite are now both initialized
```

Parameter delegates. As you write multiple programs, you might find yourself wanting to reuse existing arg classes, which is something you can do with parameter delegates. In short, parameter delegates are pointers to other arg classes. In the previous example, I decided to create two different arg classes and declare these directly in JCommander; but instead, I might want to delegate to them. This is done with the `@ParameterDelegate` annotation:

```
class MainParams {
  @Parameter(names = "-v")
  private boolean verbose;

  @ParametersDelegate
  private ArgRead argRead = new ArgRead();

  @ParametersDelegate
  private ArgWrite argWrite = new ArgWrite();
}
```

After this declaration, I need to declare one argument parameter, `MainParams`, and it will contain the aggregation of both `ArgRead` and `ArgWrite`.

Polyglotism

Thanks to the JVM's ability to support multiple languages, JCommander is trivial to use from any JVM language. I'm currently using it on a Kotlin project:

```
class Args {
    @Parameter(names = arrayOf("--buildFile"))
    var buildFile: String? = null

    @Parameter(names = arrayOf("--tasks"))
    var tasks: Boolean = false
}

fun main(argv: Array<String>) {
    val args = Args()
    JCommander(args).parse(*argv)
    println("Args: ${args}"")
}
```

Here is an example in Groovy:

```
import com.beust.jcommander.*

class Args {
  @Parameter(names = ["-f", "--file"],
      description = "File to load.")
  List<String> file
}

new Args().with {
  new JCommander(it, args)
  file.each {
    println "file: ${new File(it).name}"
  }
}
```

And here is the same example in Scala:

```
import java.io.File
import com.beust.jcommander.JCommander
import com.beust.jcommander.Parameter
import collection.JavaConversions._

object Main {
  object Args {
    @Parameter(
      names = Array("-f", "--file"),
      description = "File to load.")
    var file: java.util.List[String] = null
  }

  def main(args: Array[String]): Unit = {
    new JCommander(Args, args.toArray: _*)
    for (filename <- Args.file) {
      val f = new File(filename)
      printf("file: %s\n", f.getName)
    }
  }
}
```

Conclusion

JCommander has many other features, including the following:

- Internationalization, so your description texts can be properly localized
- Parameter hiding
- Allowing abbreviated options
- Optional case insensitivity
- Default values and default value factories
- Dynamic parameters (parsing parameters that are not known at compile time)

In sum, JCommander is a flexible library for parsing command-line parameters that also assists you in making your parsing and interpreting easy to maintain and evolve. </article>

LEARN MORE

- JCommander on GitHub
- JCommander discussion group
- JCommander example file

FABIAN LANGE

BIO

Runtime Code Generation with Byte Buddy

Create agents, run tools before main() loads, and modify classes on the fly.

An often overlooked feature of the Java platform is the ability to modify a program's bytecode before it is executed by the JVM's interpreter or just-in-time (JIT) compiler. While this capability is used by tools, such as profilers and libraries that do object-relational mapping, it is rarely used by application developers. This represents untapped potential, because generating code at runtime allows for easy implementation of cross-cutting concerns such as logging or security, changing the behavior of third-party libraries—sometimes in the form of mocking—or writing performance data collection agents.

Unfortunately, generating bytecode at runtime has been difficult until recently. There are presently three major libraries for generating bytecode:

- ASM
- cglib
- Javassist

These libraries were all designed to write and modify specific bytecode instructions from Java code. But to be able to use them, you need to understand how bytecode works, which is quite different than understanding Java source code. In addition, these libraries are harder to use and test than Java code, because the Java compiler cannot verify whether, for example, the argument order of a method call matches its signature or whether it violates the Java Language Specification. Lastly, due to their age, these libraries do not all support the new Java features, such as annotations, generics, default methods, and lambdas.

The following example illustrates how you would implement a method that calls another static method with a single string parameter using the ASM library:

```
methodVisitor.visitVarInsn(Opcodes.ALOAD, 0);
methodVisitor.visitMethodInsn(
  Opcodes.INVOKESTATIC
  "com/instana/agent/Agent"
  "record"
  "(Ljava/lang/String;)V"
)
```

cglib and Javassist are not much different. They all require usage of bytecode and String representation of signatures, which as you can see looks more like assembly language, rather than Java.

Byte Buddy is a new library that takes a different approach to solving this problem. Byte Buddy's mission is to make runtime code generation accessible to developers who have little to no knowledge of Java instructions. The library also aims to support all Java features, and is not limited to generating dynamic implementations for interfaces, which is the approach used in the JDK's built-in proxy utilities. The Byte Buddy API abstracts away all bytecode operators behind plain old Java method calls. However, it retains a backdoor to the ASM library, on top of which Byte Buddy is implemented.

Note: All the examples in this article are using the 0.6 API of Byte Buddy.

Hello World, Byte Buddy

The following HelloWorld example from the Byte Buddy documentation (see Listing 1) presents everything you need to create a new class at runtime in a concise way.

Listing 1.

```
Class<? extends Object> clazz = new ByteBuddy()
  .subclass(Object.class)
  .method(ElementMatchers.named("toString"))
  .intercept(FixedValue.value("Hello World!"))
  .make()
  .load(getClass().getClassLoader(),
        ClassLoadingStrategy.Default.WRAPPER)
  .getLoaded();
assertThat(clazz.newInstance().toString(),
           is("Hello World!"));
```

All of Byte Buddy's APIs are builder-style fluent APIs supporting the functional style. You start off by telling Byte Buddy which class you want to subclass. While in this example you simply subclass `Object`, you could subclass any non-final class, and Byte Buddy will ensure that the generic return type will be `Class<? extends SuperClass>`. Now that you have a builder for your subclass, you can tell Byte Buddy to intercept calls to a method that is named `toString` and return a fixed value instead of calling the method that is already defined by `java.lang.Object`.

You might wonder about the term *intercept* here. Usually when you subclass something, you typically use the term *override* when you change the implementation of a superclass method in a subclass. *Intercept* is a term from aspect-oriented programming (AOP), which describes a more powerful concept of "what to do" when a method is called.

After you finish declaring how the subclass behaves, you invoke `make` to get a so-called `Unloaded` representation of your class. This representation behaves like a .class file and, in fact, it even supports functions to store the class file.

Finally, as shown in Listing 1, you load the class using a class loader and get a reference to the loaded class. When getting started with Byte Buddy, the `ClassLoadingStrategy` used to do this does not usually matter. However, there are situations in which you need a specific class loader to load the new class for visibility purposes or for enforcing a specific loading order.

Note that a class generated by Byte Buddy is indistinguishable from regular classes. Unlike other libraries or proxies, there are no traces left behind. The generated code fully resembles the code that a Java compiler would create for implementing such a subclass.

ElementMatchers and Implementations

When you use Byte Buddy to add or change behavior of classes, the most common task is to look up fields, constructors, and methods. To ease these tasks Byte Buddy comes with plenty of useful predefined ElementMatchers, such as `hasParameter()` and `isAnnotatedWith()`, which check the method signature. It also has convenience aliases such as `isEquals()` and `isSetter()`, which use common Java naming patterns to match the method name. Using the predefined matchers allows for a concise description of the methods to intercept, which would otherwise be quite verbose to write. Additionally, it is possible to implement a custom ElementMatcher to cover any more complex use case.

Additionally, there exist many predefined replacement Implementations to be used in `intercept()`. Two examples are `MethodCall`, which can invoke a different method using parameters, and `Forwarding`, which uses the identical parameters to call the same method on another object.

An even more powerful interception mechanism is represented by MethodDelegation: When delegating to a method, you can first execute your custom code, and then delegate the call to the original implementation. Additionally, you can also dynamically access the information of the original call site using the @Origin annotation, as shown in **Listing 2**. When delegating to other methods, you can also dynamically access the information of the original call site, as shown next.

Listing 2.

```
public static class Agent {
  public static String record(@Origin Method m) {
    System.out.println(m + " called");
  }
}

Class<?> clazz = new ByteBuddy()
  .subclass(Object.class)
  .method(ElementMatchers.isConstructor())
  .intercept(MethodDelegation
    .to(Agent.class)
    .andThen(SuperMethodCall.INSTANCE))
  // & make instance;
```

MethodDelegation automatically looks up the best match of method signatures in case multiple interception targets are available. While the lookup is powerful and can be customized, I recommend keeping the lookup simple and understandable. After the method has been invoked, the original call continues, thanks to andThen(SuperMethodCall.INSTANCE).

The target method can take a couple of annotated parameters. To access the arguments of the originating method, you can use @Argument(position) or @AllParameters. To obtain information about the originating method itself, you can use @Origin. The type of that parameter can be java.lang.reflect.Method, java.lang.Class, or even java.lang.invoke.MethodHandle (the latter, if used with Java 7 or later). These arguments provide information about where the method has been called from, which could be useful for debugging, or even about taking different code paths, in the event that the same method is an interception target for multiple methods.

To call the originating method or its super method from the target method, Byte Buddy provides @DefaultCall and @SuperCall parameters.

Mocking

Sometimes you want to write a unit test for a scenario that can happen at runtime, but you cannot provoke that scenario reliably for the purpose of the test, if at all. For instance, in **Listing 3**, the random number generator needs to produce a specific result for you to test the control flow.

Listing 3.

```
public class Lottery {
  public boolean win() {
    return random.nextInt(100) == 0;
  }
}

Random mockRandom = new ByteBuddy()
  .subclass(Random.class)
  .method(named("nextInt"))
  .intercept(value(0))
  // & make instance;

Lottery lottery = new Lottery(mockRandom);
assertTrue(lottery.win());
```

Byte Buddy provides various kinds of interceptors, so writing mocks, or spies, is easy. However, for more than a few mocks, I would recommend switching to a dedicated mocking library. In fact, version 2 of the popular mocking library Mockito is currently being rewritten to be based on Byte Buddy.

So far, I have used `subclass()` to create what is essentially a subclass on steroids. Byte Buddy has two other modes of operation: `rebase` and `redefine`. Both options change the implementation of the specified class; while `rebase` maintains existing code, `redefine` overwrites it. However, these modifications come with a limitation: to change already loaded classes, Byte Buddy needs to work as a Java agent (more on that shortly).

For usage in unit testing or other special cases in which you can ensure that Byte Buddy loads a class for the first time, you can change the implementation during load. For that, Byte Buddy supports a concept called `TypeDescription`, which represents Java classes in an unloaded state. You can populate a pool of them from the (not yet loaded) classpath and modify classes before loading them. For example, I can modify the Lottery class in Listing 3, as shown in Listing 4.

Listing 4.
```
TypePool pool = TypePool.Default.ofClassPath();
new ByteBuddy()
  .redefine(pool.describe("Lottery").resolve(),
   ClassFileLocator.ForClassLoader.ofClassPath())
  .method(ElementMatchers.named("win"))
  .intercept(FixedValue.value(true))
  // & make and load;

assertTrue(new Lottery().win());
```

Note: You cannot use `Lottery.class` for the call to `describe` here, because this would load the class before Byte Buddy can rewrite it. Once a Java class is loaded, it is not normally possible to unload that class.

AOP Agent with Byte Buddy

In the following example, I create a performance monitoring and logging agent. It will intercept calls to JAX-WS endpoints and print how long the call took. Such an agent needs to follow conventions explained in the Javadoc for java.lang.instrument. It is launched using the `-javaagent` command-line argument and executed before the actual `main` method (hence, the name `premain`). Usually agents install a hook for themselves, which is triggered before the regular program loads classes. This bypasses the limitation of not being able to change loaded classes. Agents are stackable, and you can use as many as you like. Listing 5 shows the code for an agent.

Listing 5.
```
public class Agent {
  public static void premain(String args,
                             Instrumentation inst) {
    new AgentBuilder.Default()
      .rebase(isAnnotatedWith(Path.class))
        .transform((b, td) ->
          b.method(
            isAnnotatedWith(GET.class)
            .or(isAnnotatedWith(POST.class)))
          .intercept(to(Agent.class)))
        .installOn(inst);
  }

  @RuntimeType
  public static Object profile(@Origin Method m,
                               @SuperCall Callable<?> c)
                               throws Exception {
    long start = System.nanoTime();
    try {
      return c.call();
    } finally {
      long end = System.nanoTime();
      System.out.println("Call to " + m + " took "
                               + (end - start) +" ns");
    }
  }
}
```

After obtaining a default `AgentBuilder`, I tell it which classes it should `rebase`. This example will modify only classes having the `javax.ws.rs.Path` annotation. Next, I tell the builder how to `transform` those classes. In this example, the agent will intercept calls to either `GET` or `POST` annotated methods and delegate to the `profile` method. For this to work, the agent needs to be hooked into the Instrumentation using `installOn()`.

The profile method itself uses three annotations: `RuntimeType`, to tell Byte Buddy that the return type `Object` needs to be adjusted to the real return type used by the method it intercepts; `Origin`, to obtain a reference to the actual method intercepted, which is used to print its name; and `SuperCall`, to actually perform the original method call. In contrast to the previous example, I need to perform the super call myself, because I want to be able to have my code executed before and after the method call—so that I can perform the timing.

Comparing the way Byte Buddy implements method interception to the default Java `InvocationHandler`, you can see that the Byte Buddy method is much more optimized due to the fact that the interception will pass in only the required arguments, while `InvocationHandler` must fulfill the following interface:

```
Object invoke(Object proxy,
    Method method, Object[] args)
```

This benefit is especially noticeable for primitive arguments or return types, which need to be autoboxed. The additional `RuntimeType` annotation causes Byte Buddy to reduce any boxing to a minimum. Even though the JVM mostly optimizes away simple boxings, this is not always true for complex interfaces such as that of the `InvocationHandler`.

Using an Agent Without -javaagent

Using an agent to generate and modify code at runtime is a powerful technique; however, forcing the `-javaagent` argument to make it work is sometimes inconvenient. Byte Buddy comes with a handy convenience feature that uses the Java Attach API, which originally was designed to load diagnostic tooling at runtime. It attaches the agent to the currently running JVM. You need the additional byte-buddy-agent.jar file, which contains the utility class `ByteBuddyAgent`. With that, you invoke `ByteBuddyAgent.installOnOpenJDK()`, which does the same thing that starting the JVM with `-javaagent` did. The only other difference with this approach is that you do not invoke `installOn(inst)`, but rather you invoke `installOnByteBuddyAgent()`.

Conclusion

Despite the existence of dynamic proxies in the JDK and three popular, third-party, bytecode-manipulation libraries, Byte Buddy fills an important gap. Its fluent API uses generics, so you do not lose track of the actual type you are modifying, which can easily happen using other approaches. Byte Buddy also comes with a rich set of matchers, transformers, and implementations, and it enables their use via lambdas, which results in relatively concise and readable code.

As a result, Byte Buddy is fully understandable by developers who are not accustomed to reading bytecodes and working at low levels. With the upcoming version 0.7, Byte Buddy will support all the infrastructure around generic types. This way, Byte Buddy allows for easy interaction with generic types and type annotations even at runtime. As someone who writes a lot of bytecode-handling code, I both recommend and use this library. [Byte Buddy received a Duke's Choice Award at this year's JavaOne conference. *—Ed.*] </article>

LEARN MORE

- JVM Specification for Java 8
- Byte Buddy on Stack Overflow

MERT ÇALIŞKAN

BIO

jsoup HTML Parsing Library

Easily parse HTML, extract specified elements, validate structure, and sanitize content.

Today, enterprise Java web application developers use HTML in every aspect of a project. This work is made difficult at times because parsing HTML content is a tedious task. Doing so without a parser framework is a most undesirable task. Fortunately, there are a handful of Java-based HTML parsers publicly available. In this article, I will focus on one of my favorites, jsoup, which was first released as open source in January 2010. It has been under active development since then by Jonathan Hedley, and the code uses the liberal MIT license.

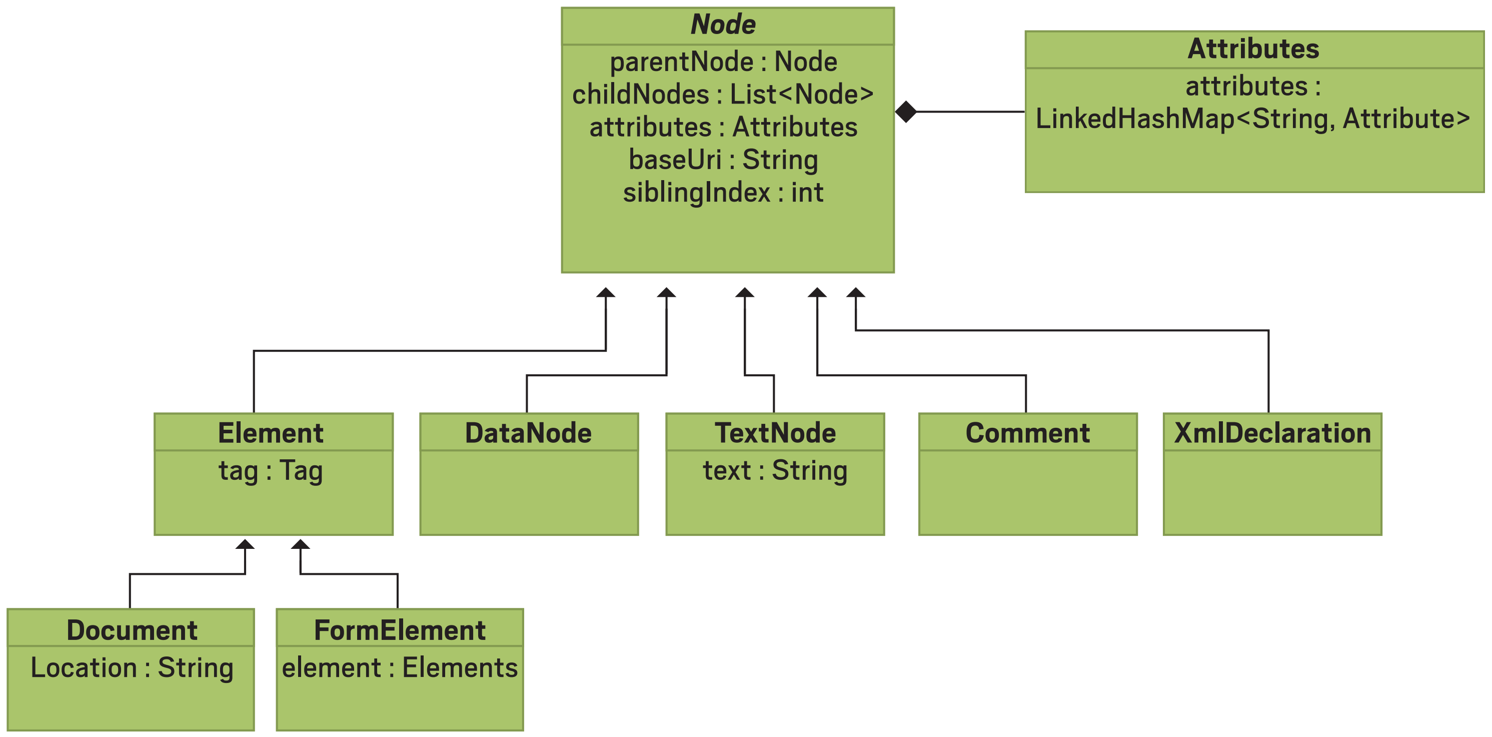


Figure 1: jsoup class diagram

What It Is

jsoup can parse HTML files, input streams, URLs, or even strings. It eases data extraction from HTML by offering document object model (DOM) traversal methods and CSS and jQuery-like selectors.

It can manipulate the content: the HTML element itself, its attributes, or its text. It also updates older content based on HTML 4.x to HTML5 or XHTML by converting deprecated tags to new versions. It can also do cleanup based on whitelists, tidy HTML output, and complete unbalanced tags automagically. I will demonstrate these features with some working examples shortly.

All the examples in this article are based on jsoup version 1.8.3, which is the latest available version at the time of this writing. The complete source code for the article is available on GitHub.

The DOM and jsoup Essentials

DOM is the language-independent representation of the HTML documents, which defines the structure and the styling of the document. **Figure 1** shows the class diagram of jsoup framework classes. Later, I'll show you how they map to the DOM elements.

The org.jsoup.nodes.Node abstract class is the main element of jsoup. It represents a node in the DOM tree, which could either be the document itself, a text node, a comment, or an element, that is, form elements, within the document. The Node class refers to its parent node and knows all the parent's child nodes.

The Element class represents an HTML element, which consists of a tag name, attributes, and child nodes. The Attributes class is a container for the attributes of the HTML elements and is composed within the Node class.

Getting Started

You can obtain the latest version of jsoup from Maven's Central Repository with the following dependency definition. Version 1.8.3 requires at least Java 5.

```
<dependency>
    <groupId>org.jsoup</groupId>
    <artifactId>jsoup</artifactId>
    <version>1.8.3</version>
</dependency>
```

Gradle users can retrieve the artifact with

```
org.jsoup:jsoup:1.8.3
```

The main access point class, org.jsoup.Jsoup, is the principal way to use the functionality of jsoup. It provides base methods that can parse either an HTML document passed to it as a file or an input stream, a string, or an HTML document provided through a URL. The example in Listing 1 parses HTML text and outputs first the node name of the element and then the text owned by the HTML element, as shown immediately below the code.

Listing 1.
```
public class Example1Main {

    static String htmlText = "<!DOCTYPE html>" +
            "    <html>" +
            "    <head>" +
            "       <title>Java Magazine</title>" +
            "    </head>" +
            "    <body>" +
            "       <h1>Hello World!</h1>" +
            "    </body>" +
            "</html>";

    public static void main(String... args) {
        Document document = Jsoup.parse(htmlText);
        Elements allElements =
            document.getAllElements();
        for (Element element : allElements) {
            System.out.println(element.nodeName()
            + " " + element.ownText());
        }
    }
}
```

The output is

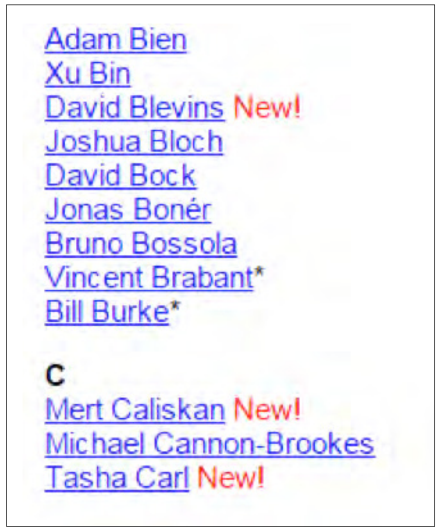
```
#document
html
head
title Java Magazine
body
h1 Hello World!
```

Ways to select DOM elements. jsoup provides several ways to iterate through the parsed HTML elements and find the requested ones. You can use either the DOM-specific getElementBy* methods or CSS

CSS and jQuery-like selectors are powerful compared with DOM-specific methods. They can be combined together to refine selection.

and jQuery-like selectors. I will demonstrate both approaches by parsing a web page and extracting all links that have HTML `<a>` tags. The code in **Listing 2** parses the Java Champions bio page and extracts the link names for all the Java Champions marked as "New!" (see **Figure 2**).

The marking was done by adding a `<font>` tag with text `New!` right next to the link. So, I will be checking for the content of the next-sibling element of each link.


Adam Bien
Xu Bin
David Blevins New!
Joshua Bloch
David Bock
Jonas Bonér
Bruno Bossola
Vincent Brabant*
Bill Burke*

C
Mert Caliskan New!
Michael Cannon-Brookes
Tasha Carl New!

Figure 2: Part of the HTML page to be parsed

Listing 2.
```
public class Example2Main {

    public static void main(String... args)
        throws IOException {
        Document document = Jsoup.connect(
            "https://java.net/website/" +
            "java-champions/bios.html" )
            .timeout(0).get();

        Elements allElements =
            document.getElementsByTag("a");
        for (Element element : allElements) {
            if ("New!".equals(
                 element.nextElementSibling()!=null
                 ? element.nextElementSibling()
                   .ownText()
                 : "")) {
                   System.out.println(
                       element.ownText());
            }
        }
    }
}
```

The same extraction of the links can also be done with selectors, as shown in **Listing 3**. It extracts the links that start with `href` value `#`.

Listing 3.
```
public class Example3Main {

    public static void main(String... args)
            throws IOException {
        Document document = Jsoup.connect
                ("https://java.net" +
            " /website/java-champions/bios.html")
            .timeout(0).get();
        Elements allElements = document.select
                ("a[href*=#]");
        for (Element element : allElements) {
            if ("New!".equals(element
                    .nextElementSibling() != null
                    ? element.nextElementSibling
                    ().ownText() : "")) {
                System.out.println(element
                        .ownText());
            }
        }
    }
}
```

Selectors are powerful compared with DOM-specific methods. They can be combined together to refine selection. In the previous code examples, we are doing the "New!" text check by ourselves, which is trivial. The example in **Listing 4** selects the `<font>` tag that contains the "New!" text, which resides after a link that has an `href` starting with the value `#`. This really shows the power of selectors.

Listing 4.
```
public class Example4Main {

    public static void main(String... args)
            throws IOException {
        Document document = Jsoup.connect
                ("https://java.net" +
            ".website/java-champions/bios.html")
            .timeout(0).get();
        Elements allElements = document.select
                ("a[href*=#] ~ font:containsOwn" +
                        "(New!)");
        for (Element element : allElements) {
            System.out.println(element
                    .previousElementSibling()
                    .ownText());
        }
    }
}
```

Here, the selectors locate the `<font>` tag as an element. I then call the `previousElementSibling()` method on it, so as to step one element back to the link. This `select()` method is available in the `Document`, `Element`, and `Elements` classes. Currently, jsoup does not support XPath queries on selectors. More information about selectors is available at the jsoup site.

Starting with version 1.6.2, jsoup supports parsing of XML files with a built-in XML parser. Note how easily this is accomplished.

Traversing nodes. jsoup provides the `org.jsoup.select.NodeVisitor` interface, which contains two methods: `head()` and `tail()`. By implementing an anonymous class from that interface and passing it as a parameter to the `document.traverse()` method, it is possible to have a callback when the node is first and last visited. The code in Listing 5 uses this technique to traverse a simple HTML text and outputs all node details.

Listing 5.
```
public class Example5Main {

    static String htmlText = "<!DOCTYPE html>" +
            "<html>" +
            "<head>" +
            "<title>Java Magazine</title>" +
            "</head>" +
            "<body>" +
            "<h1>Hello World!</h1>" +
            "</body>" +
            "</html>";

    public static void main(String... args)
            throws IOException {
        Document document = Jsoup.parse(htmlText);

        document.traverse(new NodeVisitor() {
            public void head(Node node, int depth){
                System.out.println("Node start: "
                        + node.nodeName());
            }

            public void tail(Node node, int depth){
                System.out.println("Node end: " +
                        node.nodeName());
            }
        });
    }
}
```

The output from this traversal is

```
Node start: #document
Node start: #doctype
Node end: #doctype
Node start: html
Node start: head
Node start: title
Node start: #text
Node end: #text
Node end: title
Node end: head
Node start: body
Node start: h1
Node start: #text
Node end: #text
Node end: h1
Node end: body
Node end: html
Node end: #document
```

Parsing XML files. Starting with version 1.6.2, jsoup supports parsing of XML files with a built-in XML parser. The example in Listing 6 parses an XML text and outputs it with appropriate formatting. Note once again how easily this is accomplished.

Listing 6.

```
public class Example6Main {

    static String xml =
         "<?xml version=\"1.0\"" +
         "encoding=\"UTF8\"><entries><entry>" +
         "<key>xxx</key>" +
         "<value>yyy</value></entry>" +
         "<entry><key>xxx</key>" +
         "<value>zzz</value>" +
         "</entry></entries></xml>";

    public static void main(String... args) {
        Document doc =
          Jsoup.parse(xml, "", Parser.xmlParser());
        System.out.println(doc.toString());
    }
}
```

A solution to prevent malicious HTML input is to use a WYSIWYG editor and filter the HTML output with jsoup's whitelist sanitizer.

As you would expect, the output from this is

```
<?xml version="1.0"encoding="UTF8">
<entries>
 <entry>
  <key>
   xxx
  </key>
  <value>
   yyy
  </value>
 </entry>
 <entry>
  <key>
   xxx
  </key>
  <value>
   zzz
  </value>
 </entry>
</entries>
```

It's also possible to use selectors for picking up values from specified XML tags. The code snippet in Listing 7 selects `<value>` tags that reside in `<entry>` tags.

Listing 7.

```
Document doc =
    Jsoup.parse(xml, "", Parser.xmlParser());

Elements elements = doc.select("entry value");
Iterator<Element> it = elements.iterator();
while (it.hasNext()) {
    Element element = it.next();
    System.out.println(element.nodeName() +
        " - " + element.ownText());
}
```

Preventing XSS attacks. Many sites prevent cross-site scripting (XSS) attacks by prohibiting the user from submitting HTML content or by enforcing the use of alternative markup syntax, such as markdown. An alternative solution to prevent malicious HTML input is to use a WYSIWYG editor and filter the HTML output with jsoup's whitelist sanitizer. The whitelist sanitizer parses the HTML, and iterates through it and removes the unwanted tags, attributes, or values according to the whitelist built into the framework.

jsoup also offers useful features such as tidying HTML and blending the structure or pseudo-structure of CSS selectors.

The example in Listing 8 defines a test method that cleans up HTML text according to a simple text whitelist. This list, as you will see in a moment, allows only simple text formatting with HTML tags: `b`, `em`, `i`, `strong`, and `u`.

Listing 8.
```
@Test
public void simpleTextCleaningWorksOK() {
    String html = "<div>" +
        "<a href='http://www.oracle.com'>" +
        "<b>Hello + Reader</b>!</a></div>";
    String cleanHtml = Jsoup.clean(
        html, Whitelist.simpleText());
    assertThat(cleanHtml,
               is("<b>Hello Reader</b>!"));
}
```

The `WhiteList` class offers prebuilt lists such as `simpleText()`, which limits HTML to the previous elements. There are other acceptance options, such as `none()`, `basic()`, `basicWithImages()`, and `relaxed()`.

Listing 9 shows an example of the usage of `basic()`, which allows these HTML tags: `a`, `b`, `blockquote`, `br`, `cite`, `code`, `dd`, `dl`, `dt`, `em`, `i`, `li`, `ol`, `p`, `pre`, `q`, `small`, `span`, `strike`, `strong`, `sub`, `sup`, `u`, `ul`.

Listing 9.
```
@Test
public void basicCleaningWorksOK() {
    String html = "<div><p><a " +
            "href='javascript:hackSystem()" +
            "'>Hello</a></div>";
    String cleanHtml = Jsoup.clean(html,
            Whitelist.basic());
    assertThat(cleanHtml, is("<p><a " +
            "rel=\"nofollow\">Hello</a></p>"));
}
```

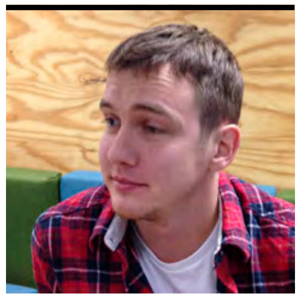
As seen in the test, the script call is eliminated and the tags that are not allowed, such as `div`, are also removed. In addition, jsoup automatically completes unbalanced tags, such as the missing `</p>` in our example.

Conclusion

In this article, I have shown only a subset of what jsoup can do. It also offers useful features such as tidying HTML, manipulating HTML tags' attributes or texts, and blending the structure or pseudo-structure of CSS selectors. Put another way, any HTML processing you might need to do is a likely candidate for using jsoup. `</article>`

LEARN MORE

- jsoup on Stack Overflow
- jsoup recipes
- Comparison of HTML parsers



OLEG ŠELAJEV

BIO

How the JVM Locates, Loads, and Runs Libraries

Class loaders are the key to understanding how the JVM executes programs.

Classes are the building blocks of Java's type system, but they also serve another fundamental purpose: a class is a compilation unit, the smallest piece of code that can be individually loaded and run a JVM process. The class-loading mechanism was set from the beginning of Java time, back in JDK 1.0, and it immensely affected Java's popularity as a cross-platform solution. Compiled Java code—in the form of class files and packaged JAR files—can be loaded into a running JVM process on any of many supported operating systems. It's this ability that has allowed developers to easily distribute compiled binaries of libraries. Because it is so much easier to distribute JAR files than source code or platform-dependent binaries, this ability has made Java popular, particularly in open source projects.

In this article, I explain the Java class-loading mechanism in detail and how it works. I also explain how classes are found in the classpath and how are they loaded into memory and initialized for use.

The Mechanics of Loading Classes into the JVM

Imagine you have a simple Java program such as the one below:

```
public class A {
 public static void main(String[] args) {
  B b = new B();
  int i = b.inc(0);
  System.out.println(i);
  }
 }
```

When you compile this piece of code and run it, the JVM correctly determines the entry point into the program and starts running the `main` method of class `A`. However, the JVM doesn't load all imported classes or even referred-to classes eagerly—that is, right away. In particular, this means that only when the JVM encounters the bytecode instructions for the `new B()` statement will it try to locate and load class `B`. Besides calling a constructor of a class, there are other ways to initiate the process of loading a class, such as accessing a static member of the class or accessing it through the Reflection API.

In order to actually load a class, the JVM uses class-loader objects. Every already loaded class contains a reference to its class loader, and that class loader is used to load all the classes referenced from that class. In the preceding example, this means that loading class `B` can be approximately translated into the following Java statement: `A.class.getClassLoader().loadClass("B")`.

Here comes a paradox: every class loader is itself an object of the java.lang.Classloader type that developers can use to locate and load the classes by name. If you're confused by this chicken-and-egg problem and wonder how the first class loader that loads all the JDK classes (for example, java.lang.String) is created, you're thinking along the right lines. Indeed, the primordial class loader, called the *bootstrap class loader*, comes from the core of the JVM and is written in native platform-dependent code. It loads the classes necessary for the JVM itself, such as those of the java.lang package, classes for Java primitives, and so forth. Application classes are loaded using the regular, user-defined class loaders written in Java—so, if needed, the developer can influence the processing of these loaders.

The class loaders in the JVM are organized into a tree hierarchy, in which every class loader has a parent. Prior to locating and loading a class, a good practice for a class loader is to check whether the class's parent can load—or already has loaded—the required class.

The Class-Loader Hierarchy

The class loaders in the JVM are organized into a tree hierarchy, in which every class loader has a parent. Prior to trying to locate and load a class, a good practice for a class loader is to check whether the class's parent in the hierarchy can load—or already has loaded—the required class. This helps avoid doing double work and loading classes repeatedly. As a rule, the classes of the parent class loader are visible to the children but are not visible otherwise. This structure, which is based on delegation and visibility of the classes, allows for separation of the responsibilities of the class loaders in the hierarchy and makes the class loaders responsible for loading classes from a specific location only.

Let's look at this hierarchy of class loaders in a Java application and explore what classes they typically load. At the root of the hierarchy, Java is the bootstrap class loader. It loads the system classes required to run the JVM itself. You can expect all the classes that were provided with the JDK distribution to be loaded by this class loader. (A developer can expand the set of classes that the bootstrap class loader will be able to load by using the `-Xbootclasspath` JVM option.)

Note that even though the library might be put on the boot classpath, it won't be automatically loaded and initialized. Classes are loaded into the JVM only on demand, so even though classes might be available for the bootstrap class loader, the application needs to access them to trigger their actual loading. (A curious aspect of this loading process is that you can override JDK classes if your JAR file is prepended to the boot classpath. While this is almost always a poor idea, it does open a door to potentially more-powerful tools.)

A sort of child of the bootstrap class loader is the *extension class loader*, which loads the classes from the extension directories (explained in a moment). These classes may be used to specify machine-specific configuration such as locales, security providers, and such. The locations of the extension directories are specified via the `java.ext.dirs` system property, which on my machine is set to the following:

```
/Users/shelajev/Library/Java/Extensions:/Library/
Java/JavaVirtualMachines/jdk1.8.0_40.jdk/Contents/
Home/jre/lib/ext:/Library/Java/Extensions:/Network/
Library/Java/Extensions:/System/Library/Java/
Extensions:/usr/lib/java
```

By changing the value of this property, you can change which additional libraries are loaded into the JVM process.

Next comes the system class loader, which loads the application classes and the classes available on the classpath. Users can specify the classpath using the `-cp` property.

Both the extension class loader and the system class loader are of the URLClassloader type and behave in the same way: delegating to the parent first, and only then finding and resolving the required classes themselves, if need dictates.

The class-loader hierarchy of web applications is a bit more complicated. Because multiple applications can be deployed simultaneously to an application server, they need to be able to distinguish their classes from each other. So, every web application uses its own class loader, which is responsible for loading its libraries. Such isolation ensures that different web applications deployed to a single server can have different versions of the same library without conflicts. So the application server automatically provides every web application with its own class loader, which is responsible for loading the application's libraries. This arrangement works because the web application class loader will try to locate the classes packaged in the application's WAR file first, rather than first delegating the search to the parent class loader.

Many security features rely on the class-loader hierarchy for permission checks.

Finding the Right Class

In general, if multiple classes with the same fully qualified name are available to the JVM, the conflict resolution strategy is simple and straightforward: the first appropriate class wins. The URLClassloader, which most of the class loaders extend from, will traverse the directories in the order they are given on the classpath and load the first class it finds that has requested the class name.

The same goes for JAR files that share the same name. The JAR files will be scanned in the order in which they appear in the classpath, not according to their names. If the first JAR file contains an entry for the required class, the class will be loaded. If not, the classpath scan will continue and reach the second JAR file. Naturally, if the class isn't found anywhere on the classpath, the `ClassNotFound` exception will be thrown.

Usually, relying on the order of directories in the classpath is a fragile practice, so instead the developer can add the classes to `-Xbootclasspath` to ensure that they will be loaded first. There's nothing in particular wrong with this approach, but maintaining a project that relies on a polluted boot classpath requires work. Intuition about where the classes are loaded from will be broken, and everyone will be confused. A better practice is to resolve the confusion at its root and figure out why there are multiple classes with the same name on the classpath. Maybe upgrading some dependency version, cleaning the caches, or running a clean build will be enough to get rid of the duplicates.

Resolution, Linking, and Verification

After a class is located and its initial in-memory representation created in the JVM process, it is verified, prepared, resolved, and initialized.

- **Verification** makes sure that the class is not corrupted and is structurally correct: its runtime constant pool is valid, the types of variables are correct, and the variables are initialized prior to being accessed. Verification can be turned off by supplying the `-noverify` option. If the JVM process does not run potentially malicious code, strict verification might not be required. Turning off the verification can speed up the startup of the JVM. Another benefit is that some classes, especially those generated on the fly by various tools, can be valid and safe for the JVM but unable to pass the strict verification process. In order to use such tools, the developer should disable this verification, which is often acceptable to do in a development environment.

- **Preparation** of a class involves initializing its static fields to the default values for their respective types. (After preparation, fields of type `int` contain 0, references are null, and so forth.)
- **Resolution** of a class means checking that the symbolic references in the runtime constant pool actually point to valid classes of the required types. The resolution of a symbolic reference triggers loading of the referenced class. According to the JVM specification, this resolution process can be performed lazily, so it is deferred until the class is used.
- **Initialization** expects a prepared and verified class. It runs the class's initializer. During initialization, the static fields are initialized to whatever values are specified in the code. The static initializer method that combines the code from all the static initialization blocks is also run. The initialization process should be run only once for every loaded class, so it is synchronized, especially because the initialization of the class can trigger the initialization of other classes and should be performed with care to avoid deadlocks.

More detail on how the JVM performs the loading, linking, and initializing of classes is explained in Chapter 5 of the Java Virtual Machine Specification.

Other Considerations About Class Loaders

The class-loading model is the central piece of the dynamic operations of the Java platform. Not only does it allow for dynamic location and linking of classes at runtime, but it also provides an interface for various tools to hook into the application.

In addition, many security features rely on the class-loader hierarchy for permission checks. For example, the famous method `sun.misc.Unsafe.getUnsafe()` successfully returns an instance of the Unsafe class if it is called from a class that was loaded by the bootstrap class loader. Because only system classes are returned by this loader, every library that uses the Unsafe API must rely on the Reflection API to read the reference from a private field.

Conclusion

When you're developing a library or a framework, as a rule, you don't have to worry about any issues with class loading. It is a dynamic process that happens at runtime, so you rarely need to influence it. Also, modifying the class-loading scheme rarely benefits a typical Java library.

However, if you create a system of modules or plugins that are intended to be isolated from each other, enhancing the class-loading scheme might be a good idea. Just remember that custom class loaders, being a fundamental force influencing all the classes, can introduce hard-to-spot bugs into literally any part of your application. So take extra care when designing your own class-loading functionality.

In this article, we looked at how the JVM loads classes into the runtime, at the hierarchical model of class loaders Java uses, and the hierarchy model of a typical Java application.

All in all, even if you don't fight class-loading issues or create plugin architectures every day, understanding class loading helps you to understand what is happening in your application. It also provides insight into how several Java tools work. And it really demonstrates the benefits of keeping your classpath clean and up to date. </article>

Oleg Šelajev (@shelajev) is an engineer, author, speaker, lecturer, and developer advocate at ZeroTurnaround. He enjoys spending time tinkering with Clojure, Git, and MacVim and is pursuing a PhD in dynamic software updates and code evolution at the University of Tartu.

LEARN MORE

- Information on controlling class loaders
- Class loaders in the JVM Specification

ANTONIO GONCALVES

BIO

Part 4

Contexts and Dependency Injection: The New Java EE Toolbox

Integration with Java EE

This series has attempted to demystify Contexts and Dependency Injection (CDI). In the previous articles, which appeared in the last three issues, I discussed what strong typing really means in dependency injection, how to use CDI to integrate third-party frameworks, and how to create loose coupling with interceptors, decorators, and events. This final article covers the integration of CDI with Java EE.

Java EE is an extension of the Java runtime. It provides a managed environment in which containers provision components with a certain number of services. These services can be lifecycle management, security, validation, persistence, or, of course, injection. Persistence and transactions are often bundled together to develop the back end of an application.

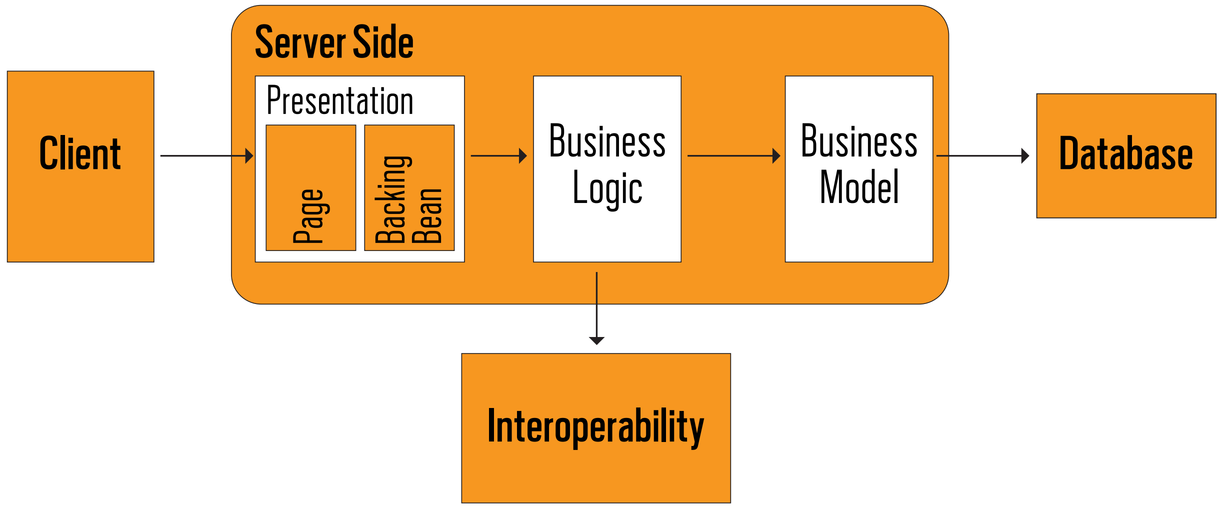


Figure 1. Standard tiers of an application

On the web tier, Java EE comes with servlets, WebSockets [See accompanying article. —*Ed.*], and JavaServer Faces (JSF), which are related to the user interface. CDI, whose workings I've explained in the last three articles, can bring the web tier and service tier together to create a homogeneous and integrated application.

Bringing the Web Tier and Service Tier Together

Java EE bundles several technologies that enable us to create any kind of architecture, including web application, REST interfaces, batch processing, asynchronous messaging, persistence, and so on. As shown in Figure 1, all these applications can be organized in several tiers: presentation, business logic, business model, or interoperating with external services. Depending on our needs, any kind of architecture is possible from stateless to stateful, from flat layered to multitiered. One problem, however, is that the web tier and service tier each has its own paradigm, its own language. Because of this, CDI is an important resource to bring them together.

Java for the service tier. Except for the web client (which uses HTML) and the database (which uses Database Definition Language), most of Java EE uses Java as its primary language, and, therefore, we find Java in most of the application tiers:

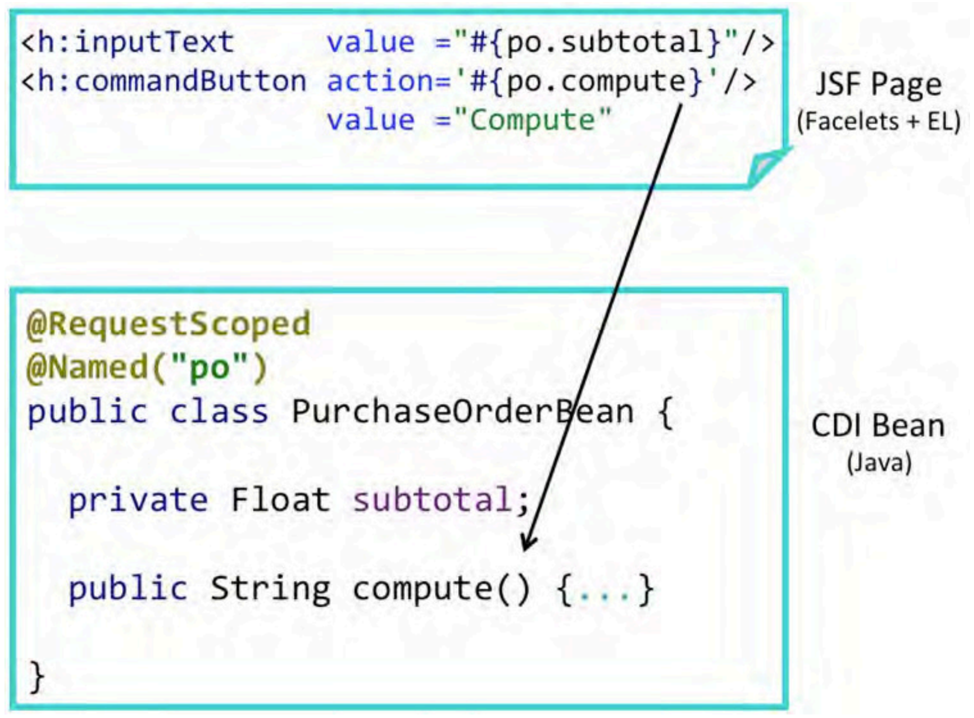
Java Persistence API entities in the business model or a simple bean on the business logic tier. We even use Java as part of our presentation tier: JSF backing beans are written in Java.

EL for the presentation tier. When I say that Java is the primary language, that's because JSF pages are written using Facelets and Expression Language, or EL. EL provides an important mechanism for enabling the presentation layer to communicate with the application logic. It is used by both JavaServer Faces technology and JavaServer Pages. It uses the # symbol. Figure 2 shows that EL uses simple expressions to dynamically access data from components—for example, where the purchase order subtotal is displayed on the page or the compute method is invoked when a button is clicked.

CDI to bind service and presentation tiers. To bind both Java and Expression Language, CDI comes to the rescue with a @Named annotation. As you can see in Figure 2, it basically gives a name to a CDI bean so the bean can be bound in EL. So here, where `PurchaseOrderBean` is annotated with `@Named("po")`, it means that the bean can be bound in EL with the name `po`.

CDI takes the concept of state management much further, applying it to the entire application, not just to the HTTP layer. Plus, CDI does this in a declarative way: by using a single annotation, the state of the bean is managed by the container.

CDI to manage state. CDI goes further by managing the state of a bean for us using scopes. Let's say that on the top right corner of our web application, we need to display the login of the user. We want this information to remain until the user's session ends. In such a case, we just annotate the bean with `@SessionScoped` and CDI will manage the state by destroying the bean when the session ends. On the other hand, computing and displaying the total of a purchase order should be done each time the page is refreshed. Because the scope of the `PurchaseOrderBean` must be shorter than the session, we can annotate it with `@RequestScoped`. CDI will maintain the state of the bean only on a per-request basis, which means that this bean is stateless. With just a few annotations, CDI unifies the web tier and service tier, eliminating glue code and letting the developer think about the business problem. CDI defines a uniform model for all our tiers bringing well-defined contexts, which is preserved across multiple requests in a user interaction.



```
<h:inputText     value ="#{po.subtotal}"/>
<h:commandButton action='#{po.compute}'/>
                 value ="Compute"
```

```
@RequestScoped
@Named("po")
public class PurchaseOrderBean {

  private Float subtotal;

  public String compute() {...}

}
```

Figure 2. Using Expression Language

Binding

Binding is the basic service for bringing together the web tier and the service tier. If we want to reference a bean in non-Java code that supports EL, such as a JSF page, we must assign the bean an EL name. The EL name is specified using the `@Named` built-in qualifier. Then we can easily use the bean in any JSF page through an EL expression. EL was originally inspired by both ECMAScript and the XPath expression languages. It was introduced in Java EE to make it easy for web page developers to access and manipulate Java in the back end without having to use JavaScript.

Expression Language. EL has a very simple syntax. It uses the hash symbol and curly brackets to identify an expression that needs to be evaluated. These expressions can be more or less complex (see Listing 1) and use arithmetic operators, lambda expressions, and so forth.

Listing 1.
```
// Value Expressions
#{purchaseOrderBean.subtotal}
#{purchaseOrderBean.customer.name}

// Array Expressions
#{purchaseOrderBean.orders[2]}

// Method Expressions
#{purchaseOrderBean.compute}

// Parameterized Method Calls
#{purchaseOrderBean.compute('5')}
```

Value expressions are the most common because they can read and write data. Here, our page would access the `subtotal` attribute or the customer `name` attribute of the `PurchaseOrderBean`. The syntax also allows access to items in an array or list, using the square bracket notation and specifying an index. As here, the expression returns the second purchase order of the bean. Another useful feature of EL is its support of method expressions. A method expression is used to invoke a public method of a bean, which can return a result. Here, the expression invokes the `compute` method of the `PurchaseOrderBean`. Parameterized method calls can use parameters. Here, the number 5 is passed as the compute value.

JSF pages. Coming back to our presentation tier, EL is present in JSF pages in different forms. In Listing 2, for example, value expressions are used to display the subtotal or value-added tax (VAT) rate of a purchase order. This binding is bidirectional, meaning that these expressions can also change the value of these attributes once the page is posted to the server. Method expressions are handy when we need to perform an action when a button is clicked, such as computing the amount of the purchase order. In this case, clicking the compute button will invoke the compute method of the `PurchaseOrderBean`.

Listing 2.
```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html">
<h:body>
    <h:form>
        <h:outputLabel value="Subtotal:"/>
        <h:inputText
          value="#{purchaseOrderBean.subtotal}"/>
        <h:outputLabel value="VAT rate:"/>
        <h:inputText
          value="#{purchaseOrderBean.vatRate}"/>
        <h:commandLink value="Compute"
          action='#{purchaseOrderBean.compute}'/>
    </h:form>
</h:body>
</html>
```

CDI beans. The `PurchaseOrderBean` in Listing 3 has `subtotal` and `vatRate` attributes, with getters and setters. It also has a compute method that is in charge of computing the total amount of the purchase order given a certain VAT rate. There is nothing special except the `@Named` annotation —without it, the bean would not have an EL name and, therefore, could not be bound to the page.

Listing 3.

```
@Named
public class PurchaseOrderBean {

    private Float subtotal = 0F;
    private Float vatRate = 5.5F;
    // …

    public String compute() {
        Float vat = subtotal * (vatRate / 100);
        Float discount =
            subtotal * (discountRate / 100);
        total = subtotal + vat - discount;
        return null;
    }
    // ...
}
```

@Named. The `@Named` annotation makes it possible to reference the bean from EL and, therefore, its attributes and methods. We can let CDI choose a name for us by not specifying an argument to the `@Named` annotation. The name defaults to the unqualified class name, decapitalized—in this case, `purchaseOrderBean` with a lowercase *p*. But we can specify an argument to the `@Named` annotation to use a nondefault name. With `@Named("order")`, the expression needs to be renamed accordingly.

Binding with Producers and Alternatives

As we've just seen, the `@Named` annotation allows the binding between an expression and a bean. Coupled with a producer, anything can then be referenced in EL. For example, we produce an integer, we name it, and it can then be referenced in an expression. Alternatives can also be used to switch the implementation not only in Java, but also in EL.

Naming a producer. To illustrate named producers and alternatives, let's take a `NumberProducer` class, the role of which is to produce numbers (see **Listing 4**). It has `vatRate` and `discountRate` attributes, both of type `Float`. The idea is to produce these attributes so they can be managed by CDI and injected somewhere else. As you know by now, this code could be ambiguous because both attributes have the same data type, `Float`. To differentiate them, we use a `@VAT` qualifier on one, and a `@Discount` qualifier on the other. Now, if we want to access the VAT rate directly on a JSF page, we just annotate the produced attribute with `@Named`. By default the EL name is `vatRate`, so the JSF page just references the `vatRate` directly, without having to prefix the name of the class: `NumberProducer (<h:inputText value="#{vatRate}"/>)`. Remember that `@Named` uses a default name that we can override. For example, instead of `vatRate` we can change the name to `vat` and reference it in this expression: `(<h:inputText value="#{vat}"/>)`.

Listing 4.

```
public class NumberProducer {

    @Produces
    @VAT
    @Named("vat")
    private Float vatRate = 5.5F;

    @Produces
    @Discount
    @Named("discount")
    private Float discountRate = 2.25f;
}
```

Alternative producer. Now, let's say we have a different use case. VAT rate and discount rate need to change depending on external configuration. For example, the VAT rate is 5.5 percent in certain countries and 19.6 percent in others, or the discount rate is usually 2.25 percent, but for Christmas it is set to 4.75 percent. This is the typical use case where alternatives can be used. First, we still need to produce, qualify, and name both the VAT and discount rate attributes (see Listing 5). Then we add an `@Alternative` annotation. As you can see, CDI is very expressive. Each annotation has its own meaning, and we can read the code very easily. Then it's just a matter of enabling or disabling the alternatives in `beans.xml`.

Listing 5.

```
public class NumberProducer {

    @Produces @VAT @Named("vat")
    private Float vatRate = 5.5F;

    @Produces
    @VAT @Named("vat") @Alternative
    private Float vatRateAlt = 19.6F;

    @Produces @Discount @Named("discount")
    private Float discountRate = 2.25f;

    @Produces
    @Discount @Named("discount") @Alternative
    private Float discountRateAlt = 4.75f;
}
```

State Management

We're all used to the concept of an HTTP session and an HTTP request. These are two examples of the broader problem of managing state that is associated with a particular context, while ensuring that all needed cleanup occurs when the context ends—for example, when the HTTP session ends, it needs to be cleaned up. Traditionally, this state management has been implemented manually, by getting and setting servlet session and request attributes. CDI takes the concept of state management much further, applying it to the entire application, not just to the HTTP layer. Plus, CDI does this in a declarative way: by using a single annotation, the state of the bean is managed by the container. No more memory leaks when the application fails to clean up session attributes; the CDI container does it automatically. CDI extends the context model defined by the Servlet specification—application, session, request—to another context: a conversation. It then applies the context to the entire business logic, not just to the web tier.

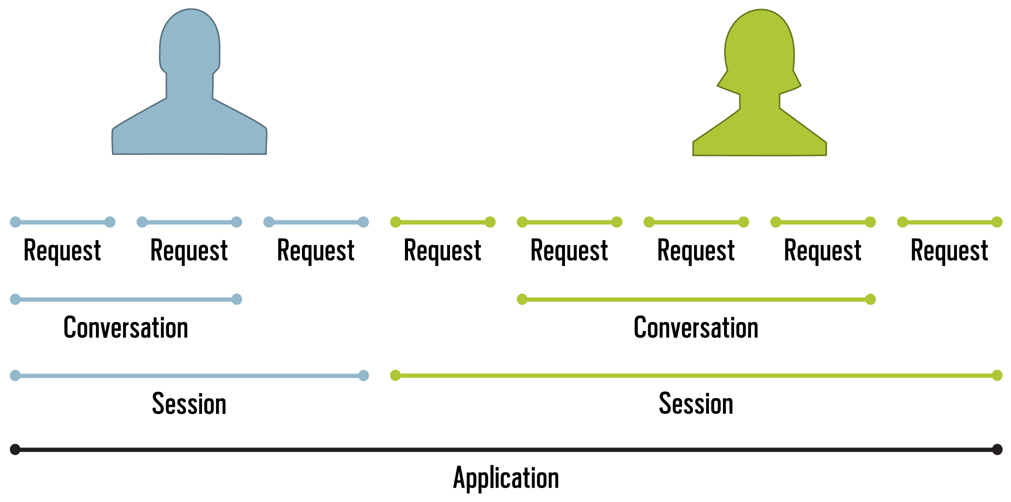


Figure 3. The four built-in CDI scopes

Built-in scope. Before looking at some code, let's first examine the four built-in CDI scopes shown in Figure 3. Let's say we have an application that has a lifespan of several months. We boot the server and leave it up and running for a few months before we shut it down. In this case, the application scope lasts for a very long time. One user logs in and remains logged in for a few minutes. The session scope spans from the moment he logs in until the moment he logs out. A second user logs in but her session stays active for a bit longer. Each session is independent and belongs to a single user, and the lifespan can be totally different. In the meantime, both users click at their own pace. Each click creates a request that is handled on the server. The last scope is the conversation and is slightly different because it can span for as long as needed. It's just a matter of beginning a conversation, which can span several requests, and ending it. Each user will have his or her own conversation. Each of these scopes is represented by an annotation.

Application scope. For example, let's say the application needs a global cache—a simple one, with just a map of key-value objects and a few methods to add data to the cache, get a value depending on the key, and remove a cache entry. We want this cache to be shared across all users' interactions within the application. For that, we just annotate the bean with `@ApplicationScoped` (see Listing 6). This cache will be automatically created by the CDI container when it is needed, and automatically destroyed when the context in which it was created ends. That is when the server is shut down. If we want this cache to be referenced directly from a JSF page, just add a `@Named` annotation.

Listing 6.
```
@Named
@ApplicationScoped
public class Cache implements Serializable {

    private Map<Object,Object> cache =
        new HashMap<>();

    public void addToCache(
        Object key, Object value) {
        // ...
    }
    public Object getFromCache (Object key) {
        // ...
    }
    public void removeFromCache (Object key) {
        // ...
    }
}
```

Session scope. Application-scope beans live during the application and are shared to all users. Session-scoped beans live during the time of the HTTP session and belong only to the current user. This scope is useful, for example, for modeling a shopping cart (see Listing 7). Each user has his own list of items and, while he's logged in, he can add items to the shopping cart and check out at the end. This instance of the shopping cart will be automatically created for the first time when the session is created and automatically destroyed when the session ends. The instance is bound to the user session and is shared by all requests that execute in the context of that session. Again, use `@Named` if invocation from EL is needed.

Listing 7.
```
@Named
@SessionScoped
public class ShoppingCart
    implements Serializable {

    private List<Item> cartItems =
        new ArrayList<>();

    public String addItemToCart() {
        // ...
    }
    public String checkout() {
        // ...
    }
}
```

Request scope. Until now, all the scopes we've covered handle state. For stateless applications, we can use the HTTP request and request scope beans. These beans usually model services (see Listing 8), or controllers, that have no state—for example, creating a book, retrieving all the book cover images, or getting a list of books depending on a category. Usually they have an `@Named` annotation because they are invoked when a button or a link on a page is clicked. An object that is defined as `@RequestScoped` is created once for every request and does not have to be serializable.

Listing 8.

```
@Named
@RequestScoped
public class BookService {

    public Book persist(Book book) {
        // ...
    }
    public List<String> findAllImages() {
        // ...
    }
    public List<Book> findByCategory(
        long categoryId) {
        // ...
    }
}
```

Conversation scope. The last built-in scope is the conversation scope. The conversation scope is a bit like the session scope in that it holds the state associated with a user and spans multiple requests to the server. However, unlike the session scope, the conversation scope is demarcated explicitly by the application. Let's say we have several web pages that form a wizard, to allow a customer to create a profile (see Listing 9). For controlling the lifecycle of a conversation, CDI gives us a `Conversation` API that may be obtained by injection. So, when a user starts to create a profile, a conversation is started by calling the begin method. The user can then go from page to page, go back to the previous page, go to the next page, and so on, until the conversation ends. As you can see, the conversation scope is the only one that needs explicit demarcation. All the other scoped beans are cleaned up by the CDI container; conversations need to be explicitly started and ended or they time out.

Listing 9.

```
@Named
@ConversationScoped
public class CustomerWizard implements
    Serializable {

    @Inject
    private Conversation conversation;

    private Customer customer =
        new Customer();

    public void initProfile () {
        conversation.begin();
        // ...
    }
    public void endProfile () {
        // ...
        conversation.end();
    }
}
```

Dependent scope. All the scopes we've just seen are contextual scopes. This means their lifecycle is managed by the container, and any injected references to the beans are also contextually aware. The CDI container ensures that the objects are created and injected at the correct time, as determined by the scope that is specified for these objects. The dependent scope is not a contextual scope and is actually called a pseudo-scope. Dependent scope is the default CDI bean scope. If a bean does not declare a specific scope, it will be injected as a dependent-scoped bean. This means that it will have the same scope as the bean where it's being injected. For example in Listing 10, if a request-scoped service (`BookService`) injects a dependent `IsbnGenerator`, then the injected `IsbnGenerator` will be request scoped. An instance of a dependent bean is strictly dependent on some other object. `IsbnGenerator` is instantiated when `BookService` is created and destroyed when `BookService` is destroyed. We can always use the `@Dependent` annotation, but we don't have to, because it is the default scope.

Listing 10.

```
@Dependent
public class IsbnGenerator {
    public String generateNumber() {
        return "13-84356-" +
            Math.abs(new Random().nextInt());
    }
}

@RequestScoped
public class BookService {

    @Inject
    private IsbnGenerator generator;
    // ...
}
```

Conclusion

In this article, we've examined how to bring the web tier and service tier together thanks to binding with the `@Named` annotation and state management with scopes. When using CDI, there is no distinction between presentation tier components and business logic components. Both can be scoped, injected, or referenced in EL. We can layer our application according to whatever architecture we need rather than being forced to bend our application logic into a technical layering. And if the architecture layering is too flat, nothing stops us from creating an equivalent layered architecture using CDI. It is possible to write Java EE applications where everything is a CDI bean. `</article>`

LEARN MORE

- CDI specification
- *Beginning Java EE 7*
- PluralSight course on CDI 1.1
- Weld CDI reference implementation

JAVAONE 2015:

Wrap-up and Review

JavaOne, the annual monster shindig for Java developers, was held this year in San Francisco, California, in late October. This year, more than 9,000 developers participated in almost 500 sessions, with the average attendee participating in 14 sessions.

The overarching themes at this year's conference were Java's 20th anniversary and its expanding presence in two areas: the cloud and the Internet of Things (IoT).

Notable among new cloud services unveiled at the show was a new one from Oracle named Java SE Cloud Service, which incorporates Java and a suite of development tools including Git, Maven, and Hudson—all aimed at moving programming to the cloud.

Java's ability to scale down to small devices was the focus of the IoT track. This theme was repeated in the Java Lounge in the vendor exhibit area, where attendees could use soldering irons and other tools to build devices using the Raspberry Pi, a small, hobbyist-oriented technology that has become wildly popular in the last few years. The Saturday before the show, JavaOne4Kids, a technical convention for children, hosted 450 attendees who learned how to program robots using Java.

An annual fixture of JavaOne is the Duke's Choice Awards, which recognize particularly meritorious Java projects or community members. The winners this year included AsciidocFX (a document creation tool), Byte Buddy (a library for generating and manipulating bytecodes, discussed in detail on page 19), OmniFaces (a library for web applications), and KumuluzEE (a microservices-enabling technology).

Oracle videotaped most of the sessions and made them available online at no cost. This carefully curated and categorized list provides an excellent way to see videos of the sessions you might have missed.

The next JavaOne conference will be held September 18–22, 2016, in San Francisco. For a listing of other conferences and events, see the Events section in this issue.

JIM BAKER AND
JOSH JUNEAU

BIO

Jython 2.7: Integrating Python and Java

A language that makes it easy to create projects with libraries from Python and Java

With an extensive community, a strong ecosystem, and a robust language, Python developers have long enjoyed high productivity. Jython is an implementation of the Python language that runs on the JVM. Reasons for choosing Jython vary widely. You might choose to use it because you want to use Java packages in your Python code; explore the Java ecosystem through Jython's interactive console; deploy a Python project that uses Django into a servlet container; or bundle your Java project with a scripting language (as found in popular tools such as Sikuli, The Grinder, and IBM WebSphere).

The first release of Jython—version 2.0—supported version 2.0 of the Python language and first saw the light of day in 2001. (There were also earlier releases under the name "JPython.") Jython has since grown to become one of the most mature and stable alternative languages for the Java platform. The most recent release, Jython 2.7, which shipped in May 2015, builds on this track record by supporting version 2.7 of the Python language, enhanced integration with Java, and extended support of the Python ecosystem, especially Python packaging.

This article gives a detailed tour of Jython 2.7 features, including an easy-to-follow example of working with spreadsheets using Apache POI. After reading this article, you will understand Jython's features enough that you can download the most recent release and get going on your own project.

A Short Primer—It's Just Python!

Jython is simply Python for the Java platform. However, if you are not familiar with the Python language, you might want a short primer.

We will use the Jython console to look at the language's features. The Jython console is powered by the popular JLine 2 project to implement a classic example of a read-evaluate-print loop (REPL). The console reads statements and expressions from the user; the statements are evaluated; and the results are printed. This process (the loop) continues until the user exits the console. Similar consoles—also powered by JLine 2—are implemented with other popular JVM languages, including Clojure, Groovy, JRuby, and Scala.

Running the Jython program without any other arguments starts up the console; we will use the most recently released version as of the writing of this article (2.7.0), as shown in Listing 1. (**Note:** In this article, we assume that you are using a UNIX-like system, with $ being the command-line prompt from a shell such as bash. Jython 2.7 also works well with Windows.)

Listing 1.
```
$ jython
Jython 2.7.0 (default:. . ., Apr 29 2015, 02:25:11)
[Java HotSpot(TM) 64-Bit Server VM (Oracle Corp.)]
```

```
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

The console prompts you with >>> for the first line of each input. Let's start by using the dict type, which is a dictionary—that is, it's a mutable mapping of keys to values. Both keys and values can be of any object: Python or Java objects, as you will see shortly. Because of their versatility, dictionaries are heavily used in most Python programs. Listing 2 demonstrates a few examples of working with dictionaries.

Listing 2.

```
>>> d1 = {'one':1, 'two':2, 'three': 3}
>>> d1['three']
3
>>> # Equivalent construction by using keywords
>>> # Note that '#' introduces a comment, including
      in the console
>>> d2 = dict(one=1, two=2, three=3)
>>> d2['one']
1
>>> d1 == d2
True
>>> len(d1)  # length of d1
3
>>> # Note that there is no construction equivalent
      using keywords,
>>> # because keywords are limited to strings that
      would also be
>>> # valid Python identifiers.
>>> inverted = {1: 'one', 2: 'two', 3: 'three'}
```

Python 2.7 also adds support for dictionary *comprehensions*. Comprehensions are no more than usages of special syntax for building specific types of collections given a generating formula. Mapping from values to keys for all the items in the original dictionary, we get the following comprehension:

```
>>> inverted_d1 =
... { v: k for k, v in d1.iteritems() }
```

(The three dots indicate that the text should be entered all on one line.) Variants of this compact comprehension syntax are also supported for making lists and sets.

This functionality in computing an inverted mapping is useful enough that we might want to do it again. So let's define a function to encapsulate it. While we can define functions in the console in Python, this is also a good chance for us to explore how to use the console in other ways.

Create a file named `basics.py`, with the following text as its initial content:

```
def inverted(d):
... return { v: k for k,v in d.iteritems() }
```

(The three dots indicate that the text should be entered on one line.) Let's look at this code fragment in some detail. First, notice that functions such as inverted are defined by using the `def` keyword. Notice that Python uses significant whitespace, instead of braces or other syntax, to describe the program's hierarchical structure. (The braces here are for the dictionary comprehension—braces mean we are constructing a `dict` or a `set` collection. Also, we will use two spaces for indentation levels in the Python code, so it fits better within the constraints of this article, although using four spaces is more typical.) Python's philosophy is simple: given that we are already indenting code so that it corresponds to its structure, it's redundant to put in braces or other syntax. But like so many formatting details, it might take a little while for you to get comfortable with Python's approach.

Let's start the Jython console again, but this time we will start the console with `jython27 -i basics.py`, which loads our file. We get the standard prompt of >>>. Let's also call the dir function to find out what is available; when the function is called with no arguments, it applies to the current module:

Jython supports Python's standard debugging and tracing mechanisms.

```
$ jython27 -i basics.py
>>> dir()
['__builtins__', '__doc__',
 '__file__',
 '__name__', '__package__',
 'inverted']
>>> inverted({1: 'one', 2: 'two'})
{'one': 1, 'two': 2}
```

The -i option means we are running the console in the scope of the basics module itself; we have a variety of defined names, as well as the inverted function we just defined. Using the REPL in this way is very useful for exploratory programming: start the console with the module in progress; try out an idea in the console; edit the module using some extract of this console work; repeat. Such exploratory programming is a hallmark of Python development.

You can also use more-traditional ways of working with your code. So IDEs such as PyDev (built on Eclipse) and PyCharm (built on IntelliJ) support GUI debugging of Python code, including breakpoints, watches, and introspecting variables. This all works because Jython supports Python's standard debugging and tracing mechanisms.

Python 2.7 introduced more functionality when working with the set type, which Jython 2.7 supports:

```
>>> {2,4,6,8,10}
set([2, 4, 6, 8, 10])
>>> # Create Empty Set
>>> set()
set([])
>>> s = {2,4,6,8,10}
>>> 3 not in s
True
```

Given that Jython supports the set type, it also supports all of the helpful Python set functionality:

```
>>> s.pop()
2
>>> s
set([4, 6, 8, 10])
>>> x.add(3)
>>> x.add(5)
>>> s.symmetric_difference(x)
set([3, 4, 5, 6, 8, 10])
```

Let's now look at Java integration. Jython is not the only way to integrate Python with Java. Other integration options include JPype (embedding CPython via JNI) and Py4J (using a remote socket connection). However, Jython is unique in how it supports working with Java objects as if they are Python objects, and vice versa.

The Python language lacks an ordered set as part of its standard library. But Jython makes it easy to use Java's available implementations of ordered sets, including java.util.LinkedHashSet, which maintains insertion ordering, and java.util.TreeSet, which maintains natural ordering. **Listing 3** demonstrates how this works.

Listing 3. (second line wraps)

```
>>> from java.util import TreeSet
>>> clangs = TreeSet(["c", "python",
    "ruby", "perl", "javascript"])
```

One nuance compared with Java is that Python does not use a new keyword to construct objects. Instead, the class itself is directly used as a factory, as shown in **Listing 4**.

Listing 4.

```
>>> jvmlangs = TreeSet(["java", "python",
"groovy", "scala", "ruby", "javascript"])

>>> clangs | jvmlangs  # set union
[c, groovy, java, javascript, perl,
python, ruby, scala]

>>> clangs & jvmlangs  # set intersection
[javascript, python, ruby]
```

Let's now try working with other Java packages. The developers of Jython are fans of the collections support in the Google Guava library, using it extensively—especially MapMaker, the library's concurrent map.

First, we need to download the Google Guava JAR file and put it in our `CLASSPATH`. (In this example, we're using Guava release 18.0.) Restart the Jython console with the `CLASSPATH` change.

Now is a good time to try out the tab completion support newly available in Jython 2.7. Sometimes, working with the Java ecosystem is a bit unwieldy: the package names are deep and often spelled out. With tab-completion support, at any point as you're typing in the console, you can press the TAB key to get a list of possible completions (if there are multiple possibilities) or the completion itself (if there is just one possibility).

So first import:

```
>>> import com.google
```

Then you can type, followed by pressing the TAB key:

```
>>> d = com.google.c
```

And you will get the following:

```
>>> d = com.google.common
```

You can eventually complete it like this:

```
>>> d = com.google.common.collect.HashBiMap.create(
... dict(one=1, two=2, three=3))
```

The advantage of this bidirectional mapping is that it is maintained over any updates, as we see here:

```
>>> d.inverse()
{3: three, 2: two, 1: one}
>>> d.inverse()[4] = "four"
>>> d
{three: 3, four: 4, two: 2, one: 1}
```

Project: Working with Spreadsheets

We are now going to explore an in-depth example to demonstrate Jython's deep support for Java integration.

Our premise is that of automating an existing business process that relies on consolidating spreadsheets to report on the state of the business. Although this spreadsheet-based process has proven its flexibility, it is also manual and error-prone. Currently this process relies on e-mail, shared drives, macros, and some proprietary tools. Sound familiar?

As software developers, we know that there are many ways this business process could be automated. We could rewrite it to remove the use of spreadsheets altogether. But we want to preserve the advantages of spreadsheets—including their wide adoption, flexibility, and ease of use. So let's try a different approach: we will continue to work with spreadsheets, but provide better tooling for managing them. We will use and integrate the following:

- Apache POI, to programmatically work with spreadsheet workbooks (Java library).
- GitHub, to version workbooks; we especially want to take advantage of its extensive REST API to store and retrieve workbooks (REST service). GitHub is, therefore, representative of general REST services we might use to manage

workbooks, including ones we might build ourselves.

- Requests, to simplify using HTTP and especially RESTful services to use GitHub's support of a REST API to resolve artifacts (Python library).
- Nosetests, to support unit testing (Python library).
- Custom Python code to glue together all of the above, including auditing and formula computation.

First, download Apache POI. As of the writing of this article, the latest version is 3.12. You will need to add JAR files from the POI distribution that support poi, poi-ooxml, poi-ooxml-schemas, and xml-beans to your `CLASSPATH`.

What makes the Python language so nice is that we can incrementally explore in a console both the problem space and possible solutions.

Next, you need to install the required Python packages. Installation support is a highlight of the Jython 2.7 release. In the past, for Python developers, one of the aggravating things about earlier releases of Jython was that they never contained full support for the Python ecosystem. Now with version 2.7, you can readily take advantage of the Python ecosystem, specifically the large number of Python packages that are available on PyPI (Python Package Index). Doing so is quite easy in Jython 2.7, because the popular pip tool is bundled with the release. This support for tooling makes it easy to incorporate the libraries and APIs of the Python ecosystem into an application.

The following command will install Nosetests and Requests modules by using the pip module. The `-m MODULE` means run the desired Python module as if it were a command-line program, passing the remaining arguments to it:

```
$ jython27 -m pip install nose requests
```

With both Java and Python dependencies now taken care of, where should we start? What makes the Python language so nice is that we can incrementally explore in a console both the problem space and possible solutions.

Assume that we have a simple spreadsheet named hours .xslx located at the top level of the GitHub repo, https://github.com/jimbaker/poi. We can retrieve this spreadsheet as https://github.com/jimbaker/poi/raw/master/hours .xlsx. Let's try this from the console, where `url` is set to the desired spreadsheet:

```
>>> import requests
>>> response = requests.get
...(url, stream=True)
```

Let's write the response to a binary file (so the file mode is `"wb"`); we will do so iteratively in 512-byte chunks so that memory consumption is minimized. The `writelines` method takes an iterator:

```
>>> f = open("hours.xlsx", "wb")
>>> f.writelines(respone.iter_content(512))
>>> f.close()
```

Now let's read the saved spreadsheet with POI; note that Jython implicitly bridges Python file objects as `FileInputStream` or `FileOutputStream` to use Java methods or constructors, as needed:

```
>>> from org.apache.poi.xssf.usermodel
... import XSSFWorkbook
>>> workbook = XSSFWorkbook(
...open("hours.xlsx", "rb"))
```

What can we do with this open workbook? Let's explore:

```
>>> dir(workbook)
```

Eventually after some exploration in the console and the POI API documentation, we might arrive at code like Listing 5 to process a workbook.

Listing 5.

```
# traverses cells in a workbook,
# calling a callback function on each cell

from contexlib import closing

def process_workbook(path, callback=None):
  if callback is None:
     def callback(cell):
       print cell,

  with open(path, "rb") as file:
    with closing(XSSFWorkbook(file)) as workbook:
      for sheet in workbook:
        for row in sheet:
          for cell in row:
            callback(cell)
```

The process_workbook function takes two parameters, path and callback. Note that callback is an optional parameter, because we give it a default value of None. However, there is no statically specified type as we would see in Java or would possibly be inferred in Scala. When we talk about static analysis of a program (or lexical analysis), it means that by only examining the program text, not running it, we—or a compiler or an IDE or some other tool—can determine certain properties of the program. What is the scope of the variables? What are the types of variables? Are types used in a consistent fashion—in other words, does the code type check? Is a chunk of code unreachable (or dead) and, therefore, can be eliminated? Can we use constant folding or inlining? And so forth. Of the items mentioned here, Jython supports only determining variable scope statically. (CPython does some amount of constant folding and dead code elimination, and Python 3.5 will have standard static type annotations as part of its support for gradual typing, a type system that mixes both dynamic and static type approaches.)

Jython enables you to move back and forth between the Java and Python ways of doing things.

We also define a function if callback is not defined, which we call, perhaps confusingly at first, callback. This function is inside the scope of the process_workbook function. But not only is it lexically scoped—and is, in fact, a closure—the callback function is conditionally defined. So that's really quite different from what we would do in Java. Once again, Python is showing that it is certainly a dynamic language; any static analysis of process_workbook could conclude only that callback might be this function. Or it might not be. However, do note that Jython has already compiled the source code to Java bytecode. So what we are seeing here is whether the name callback will be set to the corresponding compiled function body. Consequently the overhead of this conditional definition is actually no more than the overhead for some variable assignments. This shows how Jython enables you to move back and forth between the Java and Python ways of doing things.

We then iterate over the spreadsheets in each workbook, the rows in each spreadsheet, and the cells in each row. The workbook, spreadsheet, and row objects all implement java.lang.Iterable, which Jython can iterate over. Perhaps not surprisingly, Jython's integration ensures that Java code can also use for-each loops to iterate over Python iterables (and iterators).

When callback is called against cell with callback(cell), the Jython runtime does dynamic type checking: is the callback object a callable? Python has a simple rule: all callable objects implement a specific method, __call__.

All functions implement this special method, but so can arbitrary classes. Python summarizes this typing approach as *duck typing*, which is so named because if it looks like a duck, swims like a duck, and quacks like a duck, it probably is a duck. Python assumes that you know what you are doing, and lets you make the call.

Otherwise, if the `__call__` special method is not available on that specific object when the program is run, then the Python exception of `TypeError` is raised. And, of course, it's quite possible that `__call__` itself could raise an exception.

Let's now define a callback that has the ability to audit our spreadsheets for hardcoded formulas, much like an Excel formula might do. If a cell has a formula, then its formula string can be retrieved with the `getCellFormula()` method. Note that formulas in POI differ from those in Excel because they are not prefixed with an = sign.

Because Python supports properties in addition to methods, Jython further enhances how you can work with Java objects: you can treat getters and setters as if they were properties, omitting `get` and `set`, respectively. So we can write the auditing callback like so:

```
def print_if_hardcoded(cell):
  try:
    float(cell.cellFormula)
      ref = CellReference(cell)
      print ref.formatAsString(), cell
  except:
    pass
```

Here we see a common usage pattern in dynamic languages, and certainly in Python: we attempt to do something, and then catch any exceptions. (This pattern is dubbed, "it is easier to ask forgiveness than permission.") We are chaining together two accessors—retrieving the formula string (if it is not available, `IllegalStateException` is raised by POI) and then attempting to make a float value from the string (otherwise, a Python `ValueError` exception is raised). If the chain fails, we do not have a hardcoded formula after all. (These are not the formulas you're looking for.)

Assuming that the code is saved in `poi.py`, we can interactively use `jython -i poi.py`, as shown in **Listing 6**.

Listing 6.

```
>>> process_workbook(
    "example.xlsx", print_if_hardcoded)
A1 42
A2 47
```

We can readily create a script that will download workbooks with the Requests, apply this auditing, and store the results, possibly as a REST API.

Consolidate Workbooks

Let's look at a more involved example with POI. We need to consolidate all the spreadsheets from several workbooks into a single workbook. We could further extend this to build out consolidation formulas, provide formatting, and so forth, but that can be very complex due to what Excel spreadsheets can support.

The code in **Listing 7** (available in the download area) demonstrates one way to perform this procedure. It takes advantage of the argparse library, new in Python 2.7, to open up an arbitrary number of input workbooks, and then write out the consolidation to an output workbook. Defining a main function like this is idiomatic for Python code.

Functions such as the `process_workbook` function work well when we would like to do the same thing to all the cells in a workbook. In other cases, we would want to work with a subset of cells. Let's define a new function, `get_cells`, that works on ranges of cells that are defined by reference ranges, such as A1:G8, or unions of references, such as A1,B1,C1,D1.

```
def get_cells(sheet, ref):
  for row_idx, col_idx in referred(ref):
    row = sheet.getRow(row_idx)
    if row is not None:
      cell = row.getCell(col_idx)
      if cell is not None:
        yield cell
```

The `get_cells` function is a *generator* function: calling this function returns an iterator that successively yields values on each iteration (as marked by the `yield` keyword). So this is a very convenient way of constructing the equivalent of `java.lang.Iterator`, but without having to explicitly capture state between each invocation of the `next` method. Using generators is commonly seen in idiomatic Python code, especially because doing so simplifies incrementally working with data—particularly large data sets. Listing 8 (available in the download area) demonstrates using generators.

With `get_cells`, we can quickly compute answers to a number of queries. Let's try it out. What is the sum of the range of A1:G8? In other words, what is equivalent to =SUM (A1:G8) in the spreadsheet?

Define the following helper function `get_nums` and use the built-in function `sum`:

```
NUMERIC_CELLS = {
  Cell.CELL_TYPE_FORMULA,
  Cell.CELL_TYPE_NUMERIC }

def get_nums(cells):
  for cell in cells:
    if cell.cellType in NUMERIC_CELLS:
      yield cell.numericCellValue
```

Using the built-in function `sum`, the answer becomes simply `sum(get_nums(get_cells(spreadsheet, "A1:G8")))`.

Are any cells in the range A1:G8 using hardcoded formulas? Define a variant of the hardcoded audit function we had earlier and use the built-in function `any`:

```
def hardcoded_cells(cells):
  for cell in cells:
    try:
      float(cell.cellFormula)
      yield True
  except:
    yield False
```

The answer is `any(hardcoded_cells(get_cells (spreadsheet, "A1:G8")))`.

What we have done here is define the beginning of a high-level Python API for working with workbooks that resembles, in part, the functions that we might use in the spreadsheet itself. However, we also retain all the low-level Java POI library functionality as well.

This leads us to the last topic: are we able to ensure that our spreadsheets pass a set of compliance tests? This is a bit more involved, but let's say we set up a continuous integration service such as Jenkins to run tests on spreadsheets, perhaps as part of a GitHub pull request. How do we define and run our tests? The Python ecosystem has several good choices for testing frameworks, including in the standard library itself and `unittest`, which is an implementation of the xUnit style of testing. But the Nose testing framework, which builds on `unittest`, is justifiably popular because it is easy to use.

For example, we might want to ensure the correctness of our cross-tabulations: the sum of subtotals along a row should equal the sum of subtotals along a column, taking into account numerical precision issues, as shown in Listing 9.

Listing 9.
```
from nose.tools import assert_almost_equals
def assert_crosstab(spreadsheet, range1, range2):
       assert_almost_equals(
       sum(get_nums(spreadsheet, range1)),
         sum(get_nums(spreadsheet, range1)))
```

Once this is defined, you can write a simple test script like Listing 10.

Listing 10.

```
finance_wb = XSSFWorkbook("financials.xlsx")
main_sheet = finance_wb.sheetAt(0)

def test_financials():
  assert_crosstab(main_sheet, "A5:G5", "H1:H5")
```

Then you can run Nose to discover and run your tests. Nose follows a convention-over-configuration approach, which means it is easy to get going. The result is:

```
$jython -m nose
.
----------------------------
Ran 1 test in 0.033s

OK
```

Each dot on the second line corresponds to a test in all of the collected test files. Simply add more tests.

Don't Look Back!

Predictably, as time marches on, technologies and language features evolve. As such, there have been some important deprecations in Jython 2.7. Perhaps the most notable is that Jython 2.7 requires a minimum of Java 7. Another important note is that the installer no longer supports the use of an alternative JRE when generating Jython launchers. Use `JAVA_HOME` instead.

Jython 3.5

The Python language undergoes continued active development, along with its reference implementation. By the time you read this article, CPython 3.5 will be released. At some future point, a release of Jython 3.5 is planned, paralleling the CPython 3.5 release. It is worth pointing out that Jython 2.7 basically has the same internal runtime support and stdlib as Python 3.2. But substantial work will be required to get to Jython 3.5. One eagerly anticipated feature in Python 3.5 is optional static typing, which will enable even better Java integration in Jython.

But not so fast. Jython 2.7.x will be around for quite some time. The Jython team plans to work on 2.7.x as long as Python 2.7 is in wide use. The migration to and adoption of Python 3 has been fairly slow, partly due to the significant changes between versions 2.7 and 3.0. Because Python 2.7 is still in wide use, the Jython team plans to make time-based releases of the Jython 2.7.x line. The future releases of Jython 2.7.x will focus on performance, integration, and more. The release of Java 9 will likely improve performance with more optimization for dynamic languages.

Although there is no rush to get to Jython 3.5, it is on the docket. In fact, there is a branch of development already devoted to Jython 3.5, although it is in the early stages. Currently the target for a Jython 3.5 release is in the next two years.

Conclusion

Jython 2.7 provides a wealth of tools, enabling developers to combine two of the most popular ecosystems, Python and Java, in the same codebase. In this article, we took a look at some of the top new features of Jython 2.7, but there are plenty of other great features to explore. Download it from jython.org and watch for updates, which are planned for every six months. </article>

[This article is part of an ongoing series exploring JVM languages. In the last issue, we covered Kotlin. In the next issue, we examine Gosu, a JVM language used in industry for both front ends and back-end systems. —*Ed.*]

ARUN GUPTA

BIO

Using Docker in Java Applications

The lightweight virtualization container is fast becoming the preferred way to package and deploy Java web apps.

Software containers are enabling developers to package their applications, and underlying dependencies, in new ways that are portable and work consistently everywhere—on their machine, in production, in your data center, and in the cloud. Among portable containers, Docker has become the most widely used.

This first article in a two-part series explains the key concepts of Docker and how it works. In it, I demonstrate how to get started with Docker using Toolbox and how to verify the installation using a simple "Hello World" app. The concepts of a Docker image and how it is constructed will be explained. Packaging Java applications as a Docker image and running them as a container will help you to understand the basics. Some basic commands to inspect images and containers are also discussed. Finally, I show how to deploy a Java EE (WildFly) application using containers. The second part of this article will show how to create applications that require multiple Docker containers, including clusters, and I'll examine integration of Docker with some other tools.

Docker Toolbox is the easiest way to get started with Docker.

What Is Docker?

An application typically requires a specific version of the operating system, JDK, application server, database server, and a few other infrastructure components. In order to provide an optimal experience, it might need binding to specific ports, a specific amount of memory, and varying configuration settings for different components. Together, the application, infrastructure components, and configuration are an *application operating system*.

An installation script that sets up an application operating system typically performs the download, installation, and configuration of the required pieces. Docker simplifies this process by creating an *image* that contains all of these components, managed as one unit. These images are then used to create runtime *containers* that run on a virtualization platform provided by Docker. These containers can be viewed as lightweight virtual machines (VMs). They're lightweight in the sense that they do not contain copies of the entire operating system; that is provided by the virtualization platform.

Docker containers offer advantages such as isolation, sandboxing, reproducibility, constraint of resources, and snapshotting, as well as several other advantages. The containers can run without the need of a hypervisor and, because they're lightweight, they can be run at a much higher density than standard VMs.

Docker has three main components:

- Images: the build component of Docker, which consists of a read-only template of the application operating system. For example, an image could be the Fedora operating system with WildFly and your Java EE application installed. You can easily create new images or update existing ones.

- Containers: a runtime representation created from images. Containers are the run component of Docker. They can be run, started, stopped, moved, and deleted. Each container is an isolated and secure application platform.
- Registry: the distribution component of Docker, where images are uploaded and downloaded. A registry can be public, such as Docker's own registry. A private registry inside your firewall can be set up as well.

How Does Docker Work?

Docker uses a client/server architecture. The Docker daemon is the server and runs on a host machine, where it does the heavy lifting of running Docker containers. The Docker client, a binary that can be installed on your machine, communicates with the Docker daemon and sends administrative commands such as a command to retrieve an image or run a container. The Docker registry is where all the images are stored. **Figure 1** shows this typical layout.

The Docker host may be colocated with the Docker client in early development stages. But it's generally moved to a separate machine for scalability reasons.

A typical workflow entails the following:

- The Docker client asks the Docker daemon to run the container for a given image.

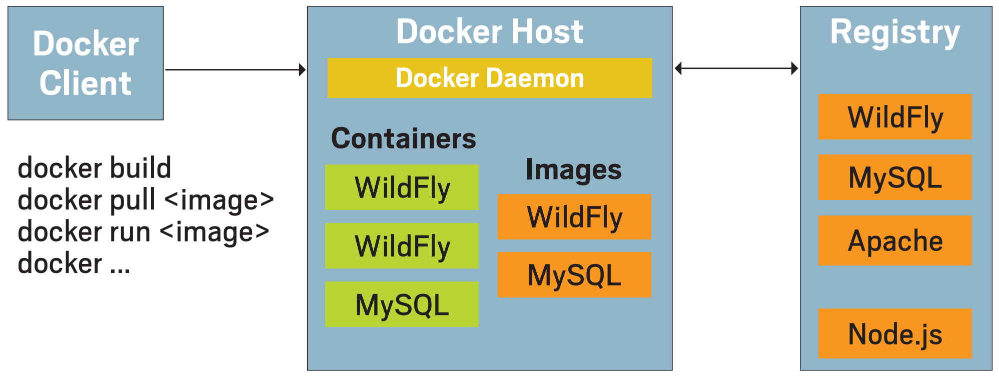


Figure 1. A typical Docker setup

- The Docker daemon checks whether the image already exists on the host. If it does, then the daemon runs the container. If not, then it downloads the image from the Docker registry and runs the container. Multiple containers that use the same image can be run easily.

The Docker client has commands for building and updating images; downloading and uploading images to a registry; running, querying, watching, and killing the running containers; and much more. It communicates with the Docker daemon using a socket or REST API. The Docker daemon talks to the Docker registry, if required, to perform the needed operation.

As a developer, you build the image for your application, run containers using that image, and then upload that image to the Docker registry for others to try it.

Getting Started with Docker

Linux machines natively support Docker, and it's easily installed using the default package manager. If you're using a Windows or Mac system, Docker Toolbox provides the different tools required to get started with Docker. It contains

- The Docker client (`docker` binary). This is the same client component previously discussed. It's used to manipulate images and containers by communicating with the Docker daemon.
- Docker Machine (`docker-machine` binary). Docker Machine lets you create Docker hosts on VMs that reside on your computer, with cloud providers, and inside your own data center. The Docker daemon is then installed inside this VM, after which a client can be configured to talk to this host.

Docker Machine creates a VM using *drivers*. A driver is an overloaded term that here means a virtual environment. For example, the Oracle VM VirtualBox driver is used on a local Mac or Windows system. AWS, Microsoft Azure, and other drivers are available to create a host in the cloud.

- Docker Compose (docker-compose binary). Often your application will consist of multiple containers—for example, WildFly, MySQL, and the Apache web server. Docker Compose enables you to define and run multi-container applications using a single configuration file.
- Kitematic, a powerful GUI for managing containers. It provides a seamless experience between the command-line interface and the GUI screens, as well as providing integration with Docker Hub.
- Docker Quickstart Terminal, a terminal application that creates a default Docker Machine and configures the Docker client to communicate with the default Docker host that is created by the Toolbox installation.
- Oracle VM VirtualBox 5.0.0, the virtualization provider for creating Docker hosts on local machines.

A Docker image is built by reading the instructions from a text file, which is usually called Dockerfile. This file contains all the commands needed to build the image.

Each of these components can be downloaded individually, but Docker Toolbox packages them nicely into a single download, and using Docker Toolbox is the easiest way to get started with Docker.

To get started, download Docker Toolbox and install it on your machine. Run Docker Quickstart Terminal to create the default Docker host and configure the Docker client to talk to this Docker host, which should generate this output:

```
Creating Machine default...
Creating VirtualBox VM...
Creating SSH key...
Starting VirtualBox VM...
Starting VM...
To see how to connect Docker to this machine,
  run: docker-machine env default
Starting machine default...
Setting environment variables for machine default..
```

This output shows that the Docker host has been created in a VirtualBox VM, the SSH keys have been created, the VM has been started, and the Docker client has been configured to talk to this Docker host. The client is configured using environment variables such as DOCKER_HOST and DOCKER_CERT_PATH. These are configured using the docker-machine env default command, as shown above. The machine name is default.

The eval $(docker-machine env default) command can be used to configure any shell to communicate with this host. Finally, it shows the following output:

```
docker is configured to use the default machine
  with IP 192.168.99.100
```

The docker-machine ip default command shows the IP address assigned to this host. It's a good idea to configure this IP address for name mapping in your /etc/hosts file or the equivalent file for your operating system. For example, you can add the following line:

```
192.168.99.100  dockerhost
```

Then run ping dockerhost to confirm that your Docker host mapping is correct. Now, the Docker client is ready to talk with the Docker host.

Docker Hello World

Before we actually run our Hello World sample using Docker, let's look at some basic commands.

docker images shows the list of images available on the host.

`docker ps` shows the list of running containers. Presently, the list of containers is empty because no containers have been started. If you also want to include previously terminated containers, then the `-a` option needs to be specified as an option. The output of this command is best seen with a width of 128 columns.

Note that `docker --help` shows the complete list of commands. Similarly, `docker ps --help` shows all the options available for a command. Feel free to check out different commands to see how you can modify them to meet your needs.

Now, let's run the prebuilt "Hello World" Docker image, which is found on Docker Hub. This is done using the following command:

```
docker run hello-world
```

The command shows the following output (some more verbiage is shown before and after this text in the output):

```
Hello from Docker.
```

The output verifies that

- The Docker client and daemon are installed correctly.
- The hello-world image is not available on the Docker daemon, but Docker was able to download the image from Docker Hub.
- The container runs from the downloaded image and streams the output to the Docker client.

As you can see, you can run your first container without much fuss.

Build Your First Docker Image Using Java

Docker images are read-only templates that launch Docker containers, and each image consists of a series of layers. Docker makes use of a *union file system* to combine these layers into a single image. Union file systems enable files and directories of separate file systems to be transparently overlaid, forming a single, coherent file system.

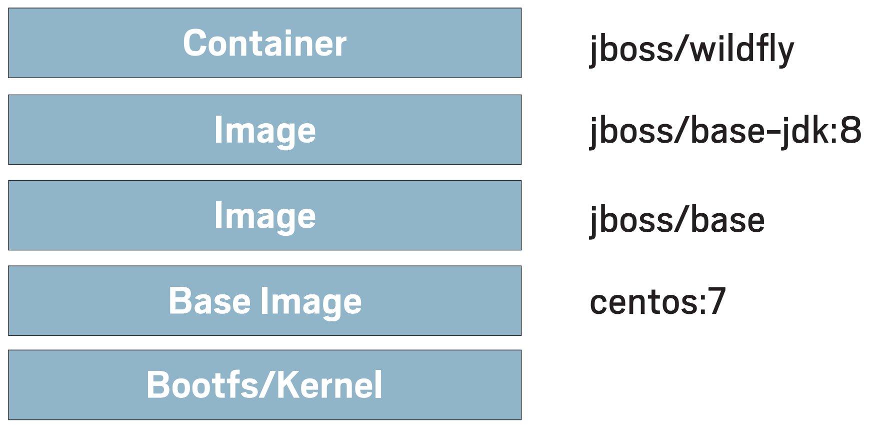


Figure 2. Building a Docker image for Java EE

This layering makes Docker extremely lightweight. Any change to a Docker image—for example, updating an application to a new version or changing the JDBC driver—requires only that the affected layer be rebuilt. Thus, rather than replacing the whole image or entirely rebuilding, as you do with a VM, only one layer is added or updated. This also makes distribution faster and simpler.

Every image starts from a base operating system image. For example, `fedora` is a base Fedora image. Multiple layers are then added. For example, `jboss/wildfly` is built using multiple images, as shown in Figure 2.

A Docker image is built by reading the instructions from a text file, which is usually called `Dockerfile`. This file contains all the commands needed to build the image. For example, it specifies the base operating system, the JDK, the application server, and any other dependencies. The JDK and application server are downloaded and installed in the image using typical shell-like commands, such as `GET`, `COPY`, and `RUN`. The `COPY` instruction can be used to copy files from the local file system into the container as well. Optionally, you might consider using a base image that already contains the JDK or WildFly and build upon that. Docker Hub has a base image of pretty much anything you need.

Each Dockerfile can have a single CMD instruction that provides the executable used for starting the container. If multiple CMD instructions are specified, then only the last CMD will take effect.

A complete reference of syntax is available online, as is a set of best practices.

A simple Dockerfile that shows only the JDK version looks like this:

```
FROM java:8
CMD ["java", "-version"]
```

Copy the content into a file and name it Dockerfile. Then, build the image:

```
docker build java-version .
```

The build command builds the Docker image using the name java-version. The . indicates that the file with the instructions to build the image is in the current directory.

Once the image has been built, it can be seen (the following shows some but not all the fields available in the output):

```
> docker images
REPOSITORY  TAG    IMAGE ID     VIRTUAL SIZE
java-sample latest 53bd2cdf4aa2 425.4 MB
```

Run the container using docker run java-sample to see the following output:

```
openjdk version "1.8.0_66-internal"
```

Using docker ps alone will not show the container in the output, because the container is not running. However, the exited container can be seen using the docker ps -a command.

If you want to run a JAR file as part of this container, you can copy the file from your local file system using COPY or you can download the JAR file using GET. Then you can include the JAR file as part of the CMD command line. Any configuration settings to the JVM can be applied here as well.

Deploy a Java EE Application Using Docker

Now that we have run a very basic example, let's see how to deploy a Java EE application to a WildFly container. Here is the Dockerfile:

```
FROM jboss/wildfly
CMD ["/opt/jboss/wildfly/bin/standalone.sh", "-c",
"standalone-full.xml", "-b", "0.0.0.0"]
RUN curl -L https://github.com/javaee-samples/
javaee7-hol/raw/master/solution/movieplex7-
1.0-SNAPSHOT.war -o /opt/jboss/wildfly/standalone/
deployments/movieplex7-1.0-SNAPSHOT.war
```

[Note the line wrapping in the last two commands. —*Ed.*] This file uses the base image of jboss/wildfly where WildFly is preinstalled in the /opt/jboss/wildfly directory. This directory is used to start a WildFly container, and the network interface is bound to all publicly available IP addresses using -b. A WAR file is downloaded from the repository and copied into the directory that is watched by WildFly for deployments.

Build this image using the following command:

```
docker build -t javaee-sample .
```

Now, let's run this image.

By default, a Docker container runs in the foreground and does not allow interaction with the terminal. The -i option allows interaction with stdin, and -t attaches a TTY (a console) to the process. Options can be combined, so -i and -t together can be specified as -it.

Run the container:

```
docker run -it -p 8080:8080 javaee-sample
```

8080 is the port that is exposed by the WildFly image. This port needs to be explicitly mapped on our host using the `-p` option. In this case, the first "8080" is the port mapped on the host and the second "8080" is the port inside the container.

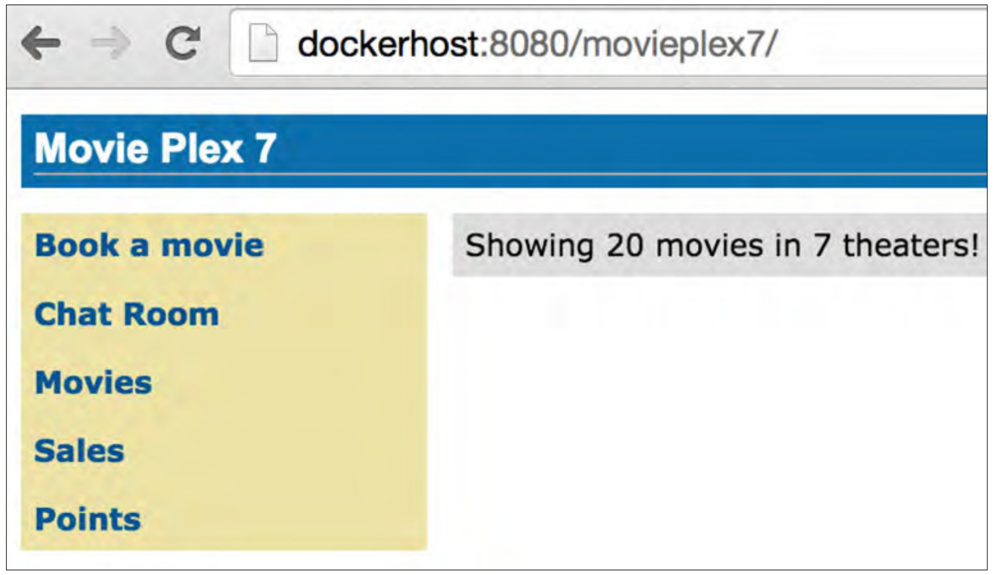
A container started this way will enable the Java EE application deployed on WildFly to be accessible on your local machine at `dockerhost:8080/movieplex7`. Note that Docker host-to-IP-address mapping was shown earlier. **Figure 3** shows the result of going to this URL.

Once the container is running in interactive mode, it can be stopped by pressing Ctrl+C. Verification that the container stopped can be seen in the following output:

```
docker ps -a

CONTAINER ID  IMAGE           COMMAND
1efa5d6f618d  jboss/wildfly  "/opt/jboss/wildfly/b"

CREATED             STATUS              PORTS
About a minute ago  Exited (130) About a minute ago

NAMES
compassionate_mestorf
```

[The output appears as a single pair of lines on a large enough screen. *–Ed.*] Each column in this output conveys meaningful information about the container:

- A unique ID assigned to each container and shown in the first column
- The image name used to start the container
- The command used to start the container
- When the container was created


dockerhost:8080/movieplex7/

Movie Plex 7

Book a movie
Chat Room
Movies
Sales
Points

Showing 20 movies in 7 theaters!

Figure 3. The sample app at port 8080

- Any ports exposed from the container (In this case, the container was already terminated, so this cell is empty.)
- The current status
- A random name that is assigned to the container, unless you specify a name using the `--name` option

The container ID is obtained on Linux/UNIX-based systems by running the following command:

```
docker ps | grep jboss/wildfly | awk '{ print $1 }'
```

The container can then be stopped using `docker stop <CONTAINER_ID>` and removed using `docker rm <CONTAINER_ID>` or stopped and removed in one step using `docker rm -f <CONTAINER_ID>`. You can also restart the container using `docker start <CONTAINER_ID>`.

Alternatively, you can run the container in a detached (background) mode by changing `-it` to `-d`.

`docker inspect` is another important command that shows more details about the running container. For example, the list of network ports inside our container can be seen easily via the following command:

```
docker inspect --format '{{ .Config.ExposedPorts }}'
<CONTAINER_ID>
```

Furthermore, the -P option can be used to map the container port to a high port (that typically is in the range of 23768 to 61000) on the local host. Then, container port-to-host port mapping can be seen:

```
docker port <CONTAINER_ID>
8080/tcp -> 0.0.0.0:32768
```

In this case, the application will be accessible at dockerhost:32768/movieplex7.

Conclusion

This article explained the key concepts of Docker and how to package your Java applications using it. Docker enables the Package Once Deploy Anywhere (PODA) paradigm, and it is changing how applications are built, deployed, and scaled. Docker reduces the impedance mismatch between development, testing, and production environments.

Ready or not, Docker is here and likely to be with us for a long time as a lightweight container technology. In the next article in this series, I'll discuss multicontainer apps and running containers in clusters. </article>

LEARN MORE

- Getting started with Docker
- Overview of containers
- Kubernetes: Docker orchestration tool

PUNE JAVA USER GROUP

Pune, with more than three million residents, is the ninth most populous city in India. It is also the second-highest software exporting city in India. Pune has been a center of learning and academia over many centuries and boasts several prestigious educational institutions.

The Pune Java User Group is one of the oldest in India. It was founded in the late 1990s and has been active ever since. The Java user group (JUG) leadership has changed several times over the years, but the focus has always been on learning new Java technologies and encouraging discussions among Java enthusiasts.

The Java language and Java EE are the usual topics of discussion. However, the group also welcomes discussions of other languages that run on the Java platform. For example, a recent talk at the JUG featured Java Champion Andres Almiray discussing the Groovy language and the build tool Gradle.

The JUG meetings are held at educational institutions and software companies in Pune. The group has more than 1,400 members and is active via multiple social media channels, such as Twitter (@JavaPune) and Google Groups. The launch events for new versions of Java have had hundreds of enthusiasts in attendance.

Pune has a buzzing startup ecosystem, and the JUG has served as a platform for interactions among tech startups. The JUG has also assisted in building tech communities in the city around many other niche as well as mainstream technologies.

Part 1

Building Apps Using WebSockets

The easy-to-use API for long-lived connections

DANNY COWARD

BIO

Java WebSockets are unlike other Java EE web components because they can push data out to web clients without clients having to ask for it. In this article, I discuss the WebSocket protocol, how WebSockets work, and how to use them in a simple project. To follow along, you need only a very basic understanding of web apps and how they operate on Java EE.

Java WebSockets are a departure from the HTTP-based interaction model, providing a way for Java EE applications to update browser and nonbrowser clients asynchronously. The interaction model for websites has long been the HTTP request/response interaction model, which is rich and allows for many sophisticated browser-based applications. Each interaction, however, always starts from the browser with some action on the part of the user: loading a page, refreshing a page, clicking a button, following a link, and so on.

For many kinds of web applications, having the user always in the driver's seat is not desirable. From financial applications with live market data, to auction applications where people around the world bid on items, to the lowly chat and presence applications, web applications have long sought means by which the server side can push data out to the client. A mix of ad hoc mechanisms arose out of this need that were either based around keeping long-lived HTTP connections or some form of client polling; none proved a complete solution to the problem. The need for a new approach led to the development of the WebSocket protocol.

The Java WebSocket API provides a set of Java API classes and Java annotations that make it relatively straightforward to create WebSocket endpoints that reside in the Java EE web container.

Introduction to the WebSocket Protocol

The WebSocket protocol is a TCP-based protocol that provides a full duplex communication channel over a single connection. In simple terms, this means that it uses the same underlying network protocol as does HTTP and that over a single WebSocket connection both parties can send messages to the other at the same time. The WebSocket protocol defines a simple connection lifecycle and a data-framing mechanism that supports binary and text-based messages. Unlike HTTP, the connections are long lived. This means that because the connection need not continually be re-established for each message transmission, as the antisymmetric HTTP protocol does, every data message in the WebSocket protocol does not need to carry all the metainformation about the connection, as HTTP does. In other words, once the connection is established, the message transmission is much lighter weight than in the HTTP protocol.

However, this is not the primary reason WebSocket is better suited to the task of servers pushing information than are polling frameworks layered on top of the HTTP protocol. Having a dedicated TCP connection to its clients makes WebSockets an inherently more efficient way for a server to update clients because data is sent only when it is needed.

To see why, imagine an online auction where 10 people are bidding on an item over a period of 12 hours. Suppose that each bidder makes an average of two successful bids on the item, so the item price changes 20 times over the period of the auction. Suppose now that the clients have to poll to see the latest bidding information. Because you cannot know when the bidders will make a bid or how up to date the amount of the current bid is, the web application supporting the auction needs to make sure that each client is refreshed at least every minute and probably more! This means that each of the clients needs to poll 60 times an hour, giving a total of 60 × 10 × 12 = 7,200 updates to make. In other words, 7,200 update messages get generated.

If, however, the server can push the data out to the client only when the data has actually changed—such as with WebSockets—only 20 messages need to be sent to each client, giving 20 × 10 = 200 messages in total.

You can probably see how the relative numbers get even more divergent as, over the lifetime of an application, either the number of clients increases or the amount of time when server data could change but doesn't increases. The server push model offered by WebSockets is inherently more efficient than any polling mechanism could ever be.

The WebSocket Lifecycle

In the WebSocket protocol, a client and a server have mostly equal roles. The only antisymmetry in the protocol is in the initial phase of the connection being established, where it matters who initiated the connection. It is somewhat like a phone call. To make the phone call happen, someone has to dial the number and someone has to answer. But once the phone call has been connected, it doesn't matter who initiated it.

For WebSockets in the Java EE platform, a WebSocket client is almost always a browser or a rich client running on a laptop, smartphone, or desktop computer, and the WebSocket server is a Java EE web application running on a Java EE application server.

Let's look now at a typical lifecycle of a WebSocket connection. First, the client initiates a connection request. This occurs when the client sends a specially formulated HTTP request to the web server. You do not need to understand every detail of the handshake request. What identifies this as a *WebSocket opening handshake request* rather than any common or garden-variety HTTP request is the use of the `Connection: Upgrade` and `Upgrade: websocket` headers, and the most important information is the request URI, `/mychat`, as shown in this handshake request:

```
GET /mychat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: megachat, chat
Sec-WebSocket-Extensions : compress, mux
Sec—WebSocket-Version: 13
Origin: http://example.com
```

The web server decides whether it supports WebSockets at all (which all Java EE web containers do) and, if so, whether there is an endpoint at the request URI of the handshake request that meets the requirements of the request. If all is well, the WebSocket-enabled web server responds with an equally specially formulated HTTP response called a *WebSocket opening handshake response*:

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

```
Sec-WebSocket-Accept: HSmrc0sM1YUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
Sec-WebSocket-Extensions: compress, mux
```

This response confirms that the server will accept the incoming TCP connection request from the client and may impose restrictions on how the connection may be used. Once the client has processed the response and is happy to accept any such restrictions, the TCP connection is created, as shown in Figure 1, and each end of the connection may proceed to send messages to the other.

Once the connection is established, several things can occur:

- Either end of the connection may send a message to the other. This may occur at any time that the connection is open. Messages in the WebSocket protocol have two flavors: text and binary.
- An error may be generated on the connection. In this case, assuming the error did not cause the connection to break, both ends of the connection are informed. Such nonterminal errors may occur, for example, if one party in the conversation sends a badly formed message.
- The connection is voluntarily closed. This means that either end of the connection decides that the conversation is over and so closes the connection. Before the connection is closed, the other end of the connection is informed of this.

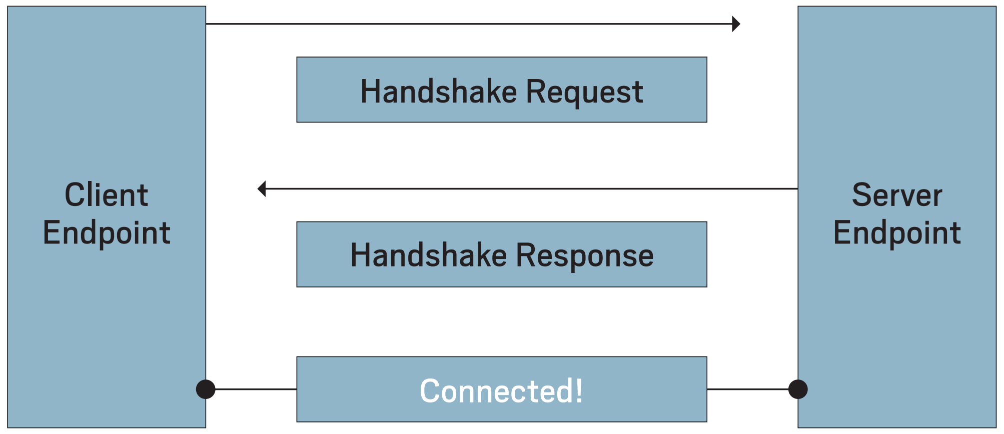


Figure 1. Establishing a WebSocket connection

Overview of the Java WebSocket API

The Java WebSocket API provides a set of Java API classes and Java annotations that make it relatively straightforward to create WebSocket endpoints that reside in the Java EE web container. The general idea is to take a Java class in which you want to implement the logic of the server endpoint and annotate it at the class level with the special Java WebSocket API annotation `@ServerEndpoint`. Next, you annotate its method with one of the lifecycle annotations, such as `@OnMessage`, which imbues the method in question with the special power of being called every time a WebSocket client sends a message to the endpoint. You then package it in the `WEB-INF/classes` directory of the WAR file. Listing 1 is an example of this.

Listing 1. The EchoServer sample

```java
import javax.websocket.OnMessage;
import javax.websocket.server.ServerEndpoint;

@ServerEndpoint{"/echo")
public class EchoServer {

    @OnMessage
    public String echo (String incomingMessage){
        return "I got this ("  +
               incomingMessage +  ")" +
               " so I am sending it back !";
    }
}
```

This WebSocket endpoint is mapped to `/echo` in the URI space of the web application. Each time a WebSocket client sends it a message, it responds back immediately with a message derived from the one it received.

The Java WebSocket API contains the means to intercept all the WebSocket lifecycle events and provides the means to send messages in both synchronous and asynchronous modes. It allows you to translate WebSocket messages to and from arbitrary Java classes using decoder and encoder classes.

The Java WebSocket API also provides the means to create WebSocket client endpoints. The only time that the WebSocket protocol is asymmetric concerns who initiates the connection. The client support in the Java WebSocket API allows a client to connect to the server, and so is suitable for Java clients to connect to WebSocket endpoints running in the Java EE web container or, in fact, any WebSocket server endpoint.

Before we look at a real example of a Java WebSocket, let's take a tour of the annotations and main classes in the Java WebSocket API. Don't worry about spending too long before we get to working code: the Java WebSocket API is one of the smaller APIs of the Java EE platform.

WebSocket Annotations

The Java WebSocket annotations have two main purposes. First, they allow you to declare that you want an ordinary Java class to become a WebSocket endpoint, and second, they allow you to annotate methods on that class so that they intercept the lifecycle events of the WebSocket endpoint. First, we will take a look at the class-level annotations.

@ServerEndpoint. This is the workhorse annotation of the API, and if you create many WebSocket endpoints, you will be seeing a lot of it. The only mandatory attribute of this class-level annotation is the value attribute (see Table 1), which specifies the URI path to which you want this endpoint to be registered in the URI space of the web application.

@ClientEndpoint. The `@ClientEndpoint` annotation is used at the class level on a Java class that you wish to turn into a client endpoint that initiates connections to server endpoints. This is often used in rich client applications that connect to the Java EE web container. It has no mandatory attributes.

@ServerEndpoint and @ClientEndpoint optional attributes. As shown in Table 2, these class-level annotations have several other attributes in common that define other configuration options that apply to the WebSocket endpoint they decorate.

Now let us turn to the lifecycle annotations.

@OnOpen. This method-level annotation declares that the Java EE web container must call the method it annotates on a WebSocket endpoint whenever a new party connects to it. The method may have either no arguments or an optional `Session` parameter, where the class `javax.websocket.Session` is an API object that represents the WebSocket connection that has just opened; and/or an optional Endpoint config parameter, where `javax.websocket.EndpointConfig` is an API object representing the con-

ATTRIBUTE	FUNCTION	MANDATORY
VALUE	DEFINES URI PATH UNDER WHICH THE ENDPOINT IS REGISTERED	YES

Table 1. The attribute of @ServerEndpoint

@SERVERENDPOINT AND @CLIENTENDPOINT ATTRIBUTES	FUNCTION	MANDATORY
CONFIGURATOR	THE CLASS NAME OF A SPECIAL CLASS THE DEVELOPER MAY PROVIDE TO DYNAMICALLY CONFIGURE THE ENDPOINT	NO
DECODERS	LIST OF CLASSES USED TO CONVERT INCOMING WEBSOCKET MESSAGES INTO JAVA CLASSES THAT REPRESENT THEM	NO
ENCODERS	LIST OF CLASSES USED TO CONVERT JAVA CLASSES INTO OUTGOING WEBSOCKET MESSAGES	NO
SUBPROTOCOLS	LIST OF STRING NAMES DENOTING ANY SPECIAL SUBPROTOCOLS, SUCH AS "CHAT," THAT THE ENDPOINT SUPPORTS	NO

Table 2. Attributes of class-level annotations

figuration information for this endpoint; and an optional WebSocket path parameter, which we will soon discuss.

@OnMessage. This method-level annotation declares that the Java EE web container must call the method it decorates whenever a new message arrives on the connection. The method must have a certain type of parameter list, but luckily, there are a number of options. The parameter list must include a variable that can hold the incoming message, can include the `Session`, and can include path parameters. A range of options exists for what kind of variables can hold the incoming message, with the most commonly used options being `String` for text messages and `ByteBuffer` for binary messages.

The method may have a specified return type or be of void return type. If there is a return type, the Java EE web container interprets the return as a message to send back immediately to the client.

@OnError. This method-level annotation declares that the Java EE web container must call the method it decorates whenever an error occurs on the connection. The method must have a `Throwable` parameter in its parameter list, and may have an optional `Session` parameter and path parameters.

@OnClose. For the final event in any WebSocket lifecycle, this method-level annotation declares that the Java EE web container must call the method it decorates whenever a WebSocket connection to this endpoint is about to close. The method is allowed to have a `Session` parameter and path parameters in its parameter list if it wants them to be passed in, as well as a `javax.websocket.CloseReason` parameter, which contains some explanation as to why the connection is closing.

The Java WebSocket API Classes

The most important API classes that the developer of Java WebSockets will encounter are the `Session`, `Remote`, and `WebSocketContainer` interfaces.

Session. The `Session` object is a high-level representation of an active WebSocket connection to an endpoint. It is available to any of the WebSocket lifecycle methods. It contains information about how the connection was established, for example, the request URI that the other party in the connection used to establish it, and the amount of time after which the connection will time out, if it's left idle. It contains the means to close the connection programmatically. It holds a map that applications may use to hold application data that they wish to associate with the connection, perhaps a transcript of the entire message that an endpoint received from a given peer. Although different from the `HttpSession` object, it is analogous in that it represents a sequence of interactions from a particular peer of the endpoint that has access to the `Session` object instance. Additionally, it holds access to the `RemoteEndpoint` interface for the endpoint.

RemoteEndpoint. The `RemoteEndpoint` interface is available from the `Session` object and represents the endpoint at the other end of the connection. In practical terms, it is the object you call when you want to send a message to the other end of the connection. There are two subtypes of `RemoteEndpoint`. One, `RemoteEndpoint.Basic`, holds all the methods for sending WebSocket messages synchronously. The other, `RemoteEndpoint.Async`, holds all the methods for sending WebSocket messages asynchronously. Many applications only send WebSocket messages synchronously because many applications have only small messages to send, so the difference between synchronous and asynchronous sending is small. Many applications send only simple text and binary messages, so knowing that the `RemoteEndpoint.Basic` interface has the following two methods will get you a long way:

```
public void sendText (String text) throws IOException;

public void sendBinary(ByteBuffer bb) throws
    IOException
```

WebSocketContainer. As the `ServletContext` is to Java servlets, so is the `WebSocketContainer` to Java WebSockets. It represents the Java EE web container to the WebSocket endpoints it contains. It holds a number of configuration properties of the WebSocket functionality, such as message buffer sizes and asynchronous send timeouts.

Let's Build Something: A WebSocket Clock

We have completed our tour of the Java WebSocket API, and having done so, we know more than enough to look at our first WebSocket application. The Clock application is a simple web application. When you run the application, you see the `index.html` web page, as shown in **Figure 2**.

When you click the Start button, the clock starts with the current time, as shown in **Figure 3**. The time updates every second.

When you click the Stop button, the clock stops until you restart it, as shown in **Figure 4**.

The application is made up of a single web page, `index.html`, and a single Java WebSocket endpoint, called `ClockServer`. When Start is pressed, `index.html` uses JavaScript code to establish a WebSocket connection with the `ClockServer` endpoint. It sends time update messages every second back to the browser client. The JavaScript code handles the incoming message and renders it on the page. Clicking Stop causes the JavaScript code in the `index.html` page to send a stop message to the `ClockServer`, which consequently stops sending the time updates. This architecture is shown in **Figure 5**.

Let's look at the code, first for the client. [The complete listing is available for download at this issue's download area. —*Ed.*] Here is the WebSocket client code from **Listing 2**.

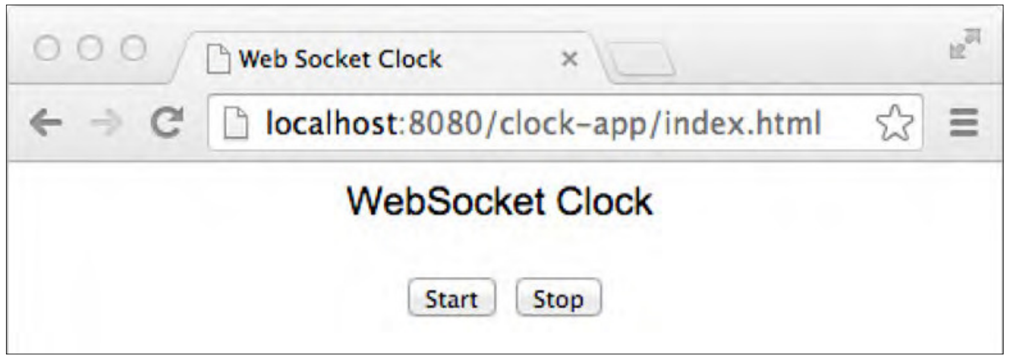


Figure 2. WebSocket Clock off

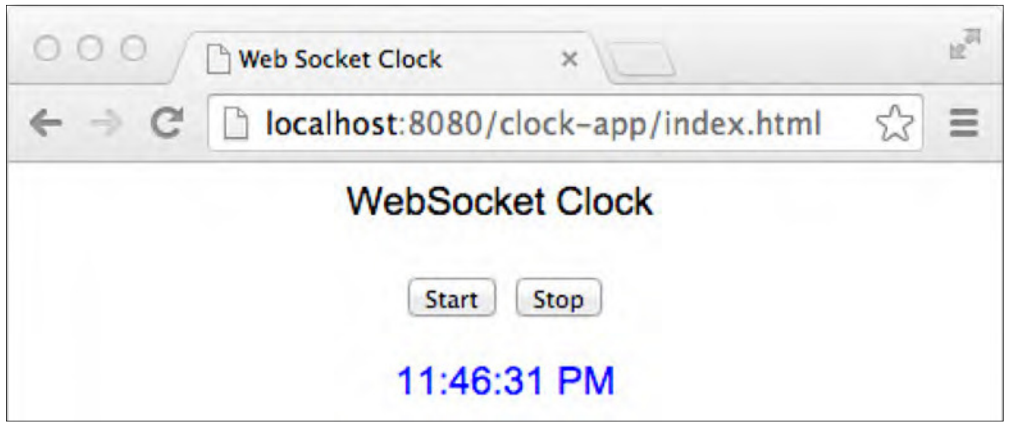


Figure 3. WebSocket Clock on

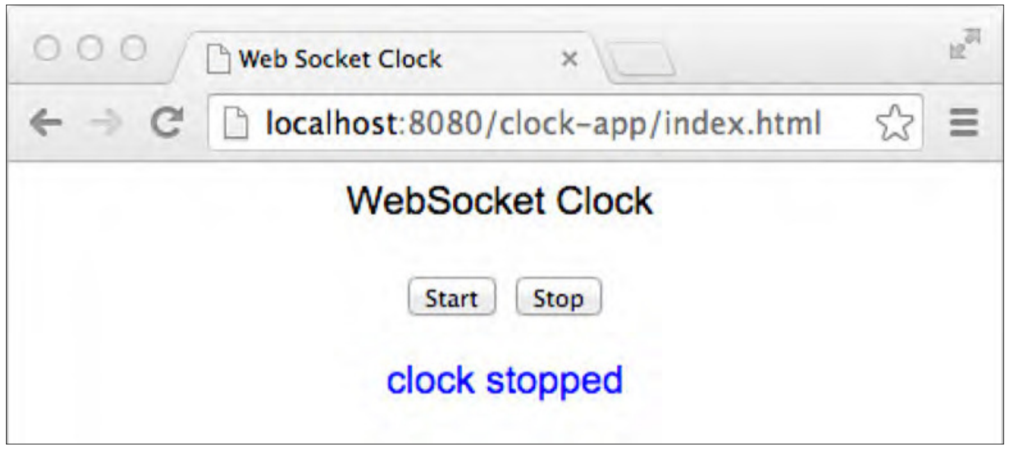


Figure 4. WebSocket Clock stopped

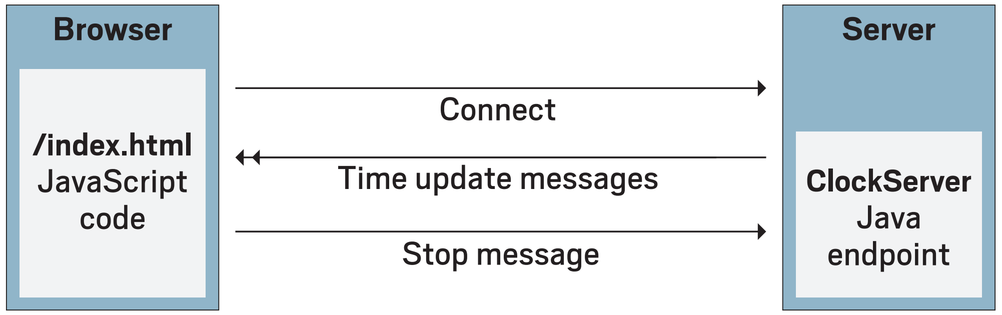


Figure 5. Clock architecture

Listing 2. WebSocket client code (JavaScript)

```
. . .
function start_clock() {
    var wsUri =
        "ws://localhost:8080/clock-app/clock";
    websocket = new webSocket(wsUri);
    websocket.onmessage = function (evt) {
        last_time = evt.data;
        writeToScreen(
           "<span style='color:  blue;'>" +
            last_time  + "</span>");
    };

    websocket.onerror = function (evt) {
        writeToScreen (
             '<span style="color:  red;"> ' +
             'ERROR:</span>  '  +  evt.data);
        websocket.close();
    };
}

function stop_c1ock() {
    websocket.send("stop");
}
```

The HTML for this page is relatively straightforward. Notice that the JavaScript API for WebSockets uses the full URI to the WebSocket endpoint where `clock-app` is the context path of the web application: `ws://localhost:8080/clock-app/clock`.

The `start_clock()` method does all the work of making the WebSocket connection and adding the event handlers, JavaScript style, particularly for handling messages that it receives from the server. The `stop_clock()` method simply sends the `stop` string to the server.

Now let's turn to the `ClockServer` endpoint, as shown in Listing 3. [Again, the complete listing is available for download at this issue's download area. —*Ed.*]

Listing 3. The server endpoint

```
...imports...

@ServerEndpoint ("/clock")
public class ClockServer {
    Thread updateThread;
    boolean running = false;

    @OnOpen
    public void startClock(Session session) {
      final Session mySession = session;
      this.running = true;
      final SimpleDateFormat sdf =
           new SimpleDateFormat("h:mm:ss a");
      this.updateThread = new Thread() {
      public void run() {
        while (running) {
          String dateString =
                 sdf.format(new Date());
          try {
            mySession.getBasicRemote().
                      sendText(dateString);
            sleep(1000);
          } catch (IOException |
                   InterruptedException ie) {
            running = false;
          }
        }
      }
      };
      this.updateThread.start();
    }

    @OnMessage
    public String handleMessage(
            String incomingMessage) {
      if ("stop".equals(incomingMessage)) {
        this.stopClock();
        return "clock stopped";
      }  else  {
          return "unknown message:   " +
                 incomingMessage;
        }
      }

      @OnError
      public void clockError(Throwable t) {
         this.stopClock() ;
      }

      @OnClose
      public void stopClock() {
        this.running = false;
        this.updateThread = null;
      }
  }
```

Notice that the ClockServer uses the @ServerEndpoint annotation to declare itself as a WebSocket endpoint, mapped to the URI/clock, relative to the context root of the web application in which it is contained. Notice that the startClock() method, called when a new client connects thanks to its @OnOpen annotation, does most of the work. It creates a thread that uses the Session object to obtain a reference to the RemoteEndpoint instance representing the client and sends it the current time, formatted into a string. If the endpoint receives a message, it is passed into the handleMessage() method, which you can identify because this method is annotated with @OnMessage. The String parameter of this method informs you that the endpoint is electing to receive its text messages in the simplest form of a Java string. This method returns a string, which is turned into a WebSocket message by the Java EE container and sent back to the client immediately.

How Many WebSocket Instances?

One question that arises even in this simple example is: how many instances will occur for a WebSocket endpoint class such as the ClockServer? The answer is that there will be one instance of the WebSocket endpoint class for each client that connects to it. Each client gets a unique endpoint instance. Further, the Java EE web container guarantees that no two WebSockets are sent to the same endpoint instance at once. So, in contrast to the Java servlet model, you can program your WebSocket endpoints knowing that there will only ever be one thread calling at a time.

Conclusion

The base WebSocket protocol gives us two native formats to work with: text and binary. This works well for very simple applications that exchange only simple information between client and server. For example, in our Clock application, the only data that is exchanged during the WebSocket messaging interaction is the formatted time string broadcast from the server endpoint and the stop string sent by the client to end the updates. But as soon as an application has anything more complicated to send or receive over a WebSocket connection, it will find itself seeking a structure into which to put the information. As Java developers, we are used to dealing with application data in the form of objects: either from classes from the standard Java APIs, or from Java classes that we create ourselves. This means that if you stick with the lowest-level messaging facilities of the Java WebSocket API and want to program using objects that are not strings or byte arrays for your messages, you need to write code that converts your objects into either strings or byte arrays and vice versa. I'll discuss this topic in the second installment of this article. </article>

This article was adapted from the book Java EE 7: The Big Picture with kind permission from the publisher, Oracle Press. The book was reviewed on page 10 of the September/October issue.

LEARN MORE

• Oracle's Java WebSockets tutorial

Quiz Yourself

Easy questions are hard for the wise man. And hard questions are easy. But what is the place for wisdom in quizzes? Let's see.

The questions in this quiz section are taken from certification test 1Z0-808: Oracle Certified Associate, Java SE 8 Programmer, Oracle Certified Java Programmer. More than last issue's questions, these little questions have unexpected traps that can snag even the attentive coder. Many times you can avoid snags by coding predictably down the middle of the language. But as we see here, even then not everything works out exactly as we expect—more so, because here we dip into a few streams, which have traps all their own. (Answers appear in the "Answers" section immediately after the questions.)

Question 1. Given these code fragments:

```
class ProductNotFoundException extends Exception{}

class SalesPerson {
  String name;
  List<String> products = new ArrayList<>();
  public List<String> getProducts() throws
    ProductNotFoundException {
        products.add("SoundCard");
           return products;
  }
}
```

and

```
class SalesApp {
    public static void main(String[] args) {
        SalesPerson sp = new SalesPerson();
        List<String> products = sp.getProducts();
        System.out.println(products.get(0));
    }
}
```

What is the result?

a. `SoundCard`
b. A `ProductNotFoundException` is thrown at runtime.
c. `0`
d. A compilation error occurs.

Question 2. Given this code fragment:

```
String wishMsg = "Happy day!";
wishMsg.concat(" Tom");
String msg = (wishMsg.length() > 10) ?
    "Too long" : "Sent";
System.out.println(msg+": "+wishMsg);
```

What is the result?

a. `Sent: Happy day!`
b. `Too long: Happy day!`
c. `Too long: Happy day! Tom`
d. A compilation error occurs.

Question 3. Given the content of the AClass.java, BClass.java, and IFace.java files:

```
public abstract class AClass {
    public void aMethod() {
        System.out.println("Method");
    }
    public abstract void bMethod();
}

public interface IFace {
    public void cMethod();
}

public abstract class BClass
    extends AClass implements IFace {}
```

Which statement is true?

a. Only the AClass.java file compiles.
b. Only the IFace.java file compiles.
c. Only the BClass.java file compiles.
d. All three files compile successfully.

Question 4. Now, let's dance along the Streams, shall we? Given this code fragment (with line numbers):

```
10. Stream<Integer> stm =
        Stream.of(10, 30, 20, 40);
11. int n1 = stm.findFirst().get();
12. boolean divByTen =
        stm.allMatch(n -> n%10 == 0);
13. System.out.println(n1 + ":" + divByTen);
```

What is the result?

a. `10:true`
b. `0:false`
c. A compilation error occurs.
d. An exception is thrown at runtime.

Question 1. Option D is correct. The program results in a compilation error. The `getProducts()` method declares that it throws `ProductNotFoundException`. `getProducts()` is invoked in the `main()` method. The `main()` method must handle `ProductNotFoundException` or must declare that it throws `ProductNotFoundException`.

Question 2. Option A is correct. Strings are immutable. `wishMsg` is unchanged and holds the value `Happy day!` Options B and C are incorrect. The result of concatenation is not retained. Option D is incorrect. The code compiles successfully.

Question 3. Option D is correct. Option A is incorrect because `AClass` compiles successfully. A class that contains a method without definition must be declared `abstract`. An `abstract` class may contain concrete methods. Option B is incorrect because the methods declared in an interface are `abstract` by default. Option C is incorrect because `BClass` does not override the `abstract` methods, `cMethod()` and `bMethod()`, and hence it is declared `abstract`.

Question 4. Option D is correct. A stream implementation may throw `java.lang.IllegalStateException` if it detects that the stream is being reused. At line 11, the `stm` stream is consumed to get the first element and it is automatically closed. At line 12, the program tried to access the elements of the closed `stm` stream to check whether the elements are divisible by 10. Therefore, the `java.lang.IllegalStateException` is thrown. If you need to traverse the same data source again, you must create a new stream. Options A and B are incorrect. A `java.lang.IllegalStateException` is thrown at runtime. Option C is incorrect. The code compiles successfully.

Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone +1.847.763.9635), who will do whatever they can to help.

Where?

Comments and article proposals should be sent to me, **Andrew Binstock**, at javamag_us@oracle.com.

While it will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A, Redwood Shores, CA 94065, USA.